MASTER EN SCIENCES INFORMATIQUES

Un analyseur syntaxique et un interpréteur abstrait pour un sous-ensemble de Java

Dony, Grégory

Award date:
2001

Facultés Universitaires Notre-Dame de la Paix
Namur
Institut d'Informatique

Un analyseur syntaxique et un interpréteur abstrait pour un sous-ensemble de Java

Grégory Dony

Mémoire présenté en vue
de l'obtention du grade de
Maître en Informatique

Promoteur : B. Le Charlier
Co-promoteur : I. Pollet

Année académique 2000-2001

Abstract

There is a strong need of static analysis in the object-oriented paradigm noticeably to get type information in order to replace dynamics methods calls by static ones. This thesis is a contribution towards the implementation of an generic analyzer of Java programs, supported by the Abstract Interpretation methodology.

This thesis consists in two main parts. The first one describes the conception and the implementation of a translator for a subset of Java into a simple language more convenient for static analysis. The second one presents a complete analyzer for a subset of this simple language.

Résumé

L'analyse statique a de nombreuses applications dans le paradigme "Orienté Objet". L'une des plus connues est l'élimination du "dynamic dispatch". Le travail effectué dans le cadre de ce mémoire s'intègre dans un projet global d'analyse statique de programmes Java. Ce projet a pour caractéristique particulière l'application directe d'une méthodologie appelée *Interprétation abstraite.*

Ce mémoire se divise en deux parties principales. La première partie décrit un "traducteur" de Java vers SAP, langage du type "syntaxe abstraite" très proche de Java mais plus simple et plus adapté à des analyses statiques. La seconde partie présente un analyseur complet pour un sous-ensemble de SAP.

Mes remerciements vont en premier lieu au Professeur B. Le Charlier et à Isabelle Pollet pour leur disponibilité et leurs précieux conseils.

Je remercie également le Professeur P. Van Hentenrijk et Laurent Michel pour leur aide lors de mon stage et leur constante attention.

Comment ne pas remercier Kik Williams et toute sa famille pour les bons moments passés ensemble.

Un merci tout particulier à mes parents qui m'ont soutenu tout au long de mes études ainsi qu'à ma grande soeur et à mon "grand frère".

Merci à Martin et à Butch pour leur soutien et leur sympathie. Merci à Cécile de m'avoir supporté pendant trois mois et demi sans jamais s'en plaindre (ou si peu).

Merci enfin à tous ceux grâce à qui ces cinq années furent tout sauf une galère.

Table des matières

Introduction générale

Il est difficile de nier le succès grandissant de la programmation orientée objet, et, en particulier, du langage Java. Cependant, on constate, d'un autre côté, que le style "Orienté Objet" induit des implémentations non efficaces. En effet, les mécanismes typiques de "l'Orienté Objet", tels l'héritage et le "dynamic dispatch", ainsi que l'objectif de réutilisation, impliquent généralement l'écriture de petites méthodes qui en utilisent, plus ou moins directement, d'autres. On se retrouve par conséquent avec des programmes contenant de nombreux appels à des méthodes virtuelles dont la version réelle est déterminée seulement au moment de l'exécution.

Pour tenter de palier à ce manque d'efficacité, on peut essayer de déterminer avant l'exécution quelles sont effectivement les méthodes appelées. Pour ce faire, il faut disposer d'une information suffisante sur le type (dynamique) des variables et ce avant l'exécution effective du programme. Ce type d'information est typiquement le résultat d'une *analyse statique*. L'analyse statique trouve donc dans le paradigme "Orienté Objet" un important champ d'application au niveau de l'optimisation du code généré (pour une étude plus détaillée des possibilités d'optimisation sur base d'analyse statique en O.O., on se référera, par exemple, à [CDG96]) .

L'analyse statique de programme O.O. peut avoir de nombreuses autres applications que celle évoquée ci-avant. On peut, par exemple, penser à des applications dans le domaine de la vérification. Ainsi, une analyse de types peut permettre la validation (ou l'invalidation) d'un cast (le casting étant source de nombreuses erreurs en programmation orientée objet).

Le travail effectué dans le cadre de ce mémoire s'intègre dans un projet global d'analyse statique de programmes Java introduit dans [Pol99]. Ce projet a pour caractérisque particulière l'application directe d'une méthodologie appelée *Interprétation Abstraite*. Cette méthodologie, introduite par P. Cousot et R. Cousot dans [CC77], offre un *framework* théorique permettant de garantir à moindre coût la correction des analyses réalisées (pour une introduction à cette méthodologie, on se réfèrera, par exemple, à [AH87, NNH99, LC92]).

Ce mémoire apporte deux contributions distinctes à ce projet et se divise donc tout naturellement en deux parties principales. La première contribution est une participation

directe à l'implémentation d'un prototype d'analyseur statique de types pour un sous-langage représentatif de Java, tandis que la seconde consiste en la réalisation complète d'un analyseur pour une restriction de ce sous-langage.

La première contribution est un "traducteur" de Java vers SAP, langage du type "syntaxe abstraite" très proche de Java mais plus simple et plus susceptible d'être facilement analysé par interprétation abstraite.

La première partie de ce mémoire est dédiée à la description de ce traducteur et des différentes étapes qui le composent. Après avoir rappelé la définition du sous-langage cible, nous expliquons tout d'abord l'analyseur syntaxique et le "type checker" que nous avons conçus et implémentés. Ensuite, nous décrivons la phase de traduction à proprement parler.

La deuxième partie consiste en l'application de l'interprétation abstraite à la vérification des types dynamiques sur un langage relativement simple. Dans un premier temps, nous introduisons succinctement la théorie de l'interprétation abstraite. Ensuite, nous définissons le langage considéré et y instancions les concepts théoriques de l'interprétation abstraite tels que présentés dans [LC99b]. Finalement, nous montrons quelques tests du logiciel réalisant cette vérification des types et nous les discutons.

Première partie

Traduction d'un programme de VTF simplifié à SAP

Introduction

La première partie du travail consiste en la réalisation d'un traducteur capable de créer, à partir d'un programme utilisant le langage VTF simplifié, un arbre exprimant ce programme dans la syntaxe abstraite SAP.

L'objectif de cette traduction est lié à certaines caractéristiques du langage SAP le rendant plus simple et plus propice aux analyses auxquelles il est destiné.

La structure de cette première partie se dessine comme suit. Tout d'abord nous définissons le langage VTF simplifié ("ce dont on part") et la syntaxe abstraite de SAP ("ce à quoi nous devons arriver"). Nous rappelons ensuite quelques principes généraux relatifs à ce genre de traduction. Enfin, nous appliquons ces principes à notre traducteur et nous décrivons les étapes successives de construction de celui-ci. Chaque étape est d'abord présentée à un niveau théorique puis, dans les grandes lignes, au niveau de son implémentation. L'implémentation a été réalisée en Java.

Chapitre 1

Définitions de VTF simplifié et SAP

VTF simplifié est un sous-ensemble du langage Java. Il est intéressant de le traiter étant donné le projet global dans lequel s'inscrit ce mémoire à savoir l'application d'une théorie particulière de vérifications des propriétés d'un programme à un langage orienté objet. Le langage SAP est lui plus proche d'un langage machine où chaque instruction fait quelque chose de simple et de toujours fini. cela, pour des raisons non évoquées ici, est plus propice à ces analyses futures.

1.1 Définition de VTF simplifié

Le langage VTF simplifié est tiré d'un premier sous-langage de Java, le langage VTF [ble...].

"Le langage VTF (Vas-T'y-Frotte) est au langage Java ce que l'île de Vas-T'y-Frotte est à l'île de Java : beaucoup plus petit." En d'autres termes, VTF est un sous-ensemble du langage Java qui se limite aux principaux aspects "orienté objet" de ce langage. Notons, entre autres différences, l'élimination de la notion d'*interface* ou encore l'absence de classes prédéfinies.

Le langage VTF simplifié connaît quant à lui trois modifications majeures par rapport à VTF.

Tout d'abord la notion de package disparaît totalement. Un programme en VTF simplifié est une simple suite de classes.

Deuxièmement, la notion d'attribut d'accessibilité n'existe plus. Tous les objets déclarés dans n'importe quelle classe sont considérés comme des objets accessibles à toutes les autres classes. Les mots clés **private**, **protected** et **public** sont donc inutiles et, s'ils

sont tolérés, ils sont simplement ignorés lors de l'analyse syntaxique.

Soulignons également l'ajout de la notion de label : au début de chaque instruction peut être ajouté un label permettant d' "identifier" cette instruction.

Enfin le type **String** a ici un statut différent de celui qu'il a en Java puisqu'il est considéré comme un type scalaire. Ce n'est peut-être pas très élégant mais c'est plutôt pratique par la suite. En effet, cela nous permet de ne pas devoir à chaque fois définir une classe **String** quand on veut utiliser une chaîne de caractères.[1]

La définition exacte de la syntaxe concrète est fournie ci-après sous la forme d'une grammaire BNF agrémentée de quelques explications.

- **Le programme**

$< program >$ ::= $< classDef >$
| $< classDef >< program >$

$< classDef >$ ::= **abstract** $< className >$ [**extends** $< className >$]
{$< abstDeclList >$**}**
concrete $< className >$ [**extends** $< className >$]
{$< concDeclList >$**}**

$< abstDeclList >$ ::= $< abstDecl >$
| $< abstDecl >< abstDeclList >$
| $< empty >$

$< abstDecl >$ ::= $< abstMethodDecl >$
| $< concDecl >$

$< concDeclList >$ ::= $< concDecl >$
| $< concDecl >< concDeclList >$
| $< empty >$

$< concDecl >$ ::= $< fieldDecl >$
| $< constructorDecl >$
| $< concMethodDecl >$

$< empty >$ ::= pas de caractère

Un programme est un suite de classes. La présence d'au moins une classe est obligatoire. Il peut s'agir de classes abstraites comme de classes concrètes.

La déclaration d'une classe est composée de son nom, d'éventuellement la classe qu'elle étend et enfin d'une liste de déclarations. Cette liste peut être vide. Une classe abstraite peut contenir des déclarations de méthodes abstraites, de constructeurs, de méthodes concrètes et des champs. Une classe abstraite peut contenir également toutes ces déclarations sauf des déclarations de méthodes abstraites.

- **Les déclarations**

[1] Ce statut du type **String** n'est pas présent dans la définition de SAP se trouvant dans [ipo...] où le type **String** est d'ailleurs tout simplement inexistant.

$< fieldDecl >$: := $< type >< fieldName >$ [= $< expression >$]
$< concMethodDecl >$: := $< type >< methodName >$ (< $paramList >$)
$< instructionBloc >$
$< abstMethodDecl >$: := **abstract** $< type >< methodName >$ (< $paramList >$) ;
$< constructorDecl >$: := $< constrName >$ (< $paramList >$) $< instructionBloc >$
$< paramList >$: := $< type >< identifier >< paramName >$
| $< type >< identifier >< paramName >$, $< paramList >$
| $< empty >$
$< instructionBloc >$: := { (< $instruction >$)* }

La déclaration d'un champ contient le nom du champ, le type du champ et éventuellement, l'affectation d'une valeur (sous la forme d'une expression quelconque).

La déclaration d'une méthode abstraite est composée de son nom, d'une liste des paramètres (peut être une liste vide) et du type résultat de la méthode. Une méthode concrète contient en plus un bloc d'instructions.

La déclaration d'un constructeur se compose du nom, d'une liste de paramètres et d'un bloc d'instructions.

- **Les types**

$< type >$: := **int** | **float** | **boolean** | className | **void**

Les types possibles sont les types flottant, entier, chaîne de caractère, booléan, void et enfin le type classe.

- **Les instructions**

$< instruction >$: := $< type >$ $< varDesignateur >$ = [$< expression >$] ;
| [$< lab >$] $varDesignateur >=< expression >$;
| [$< lab >$] [$< designateur >$] $< methodName >$ ($< exprList >$) ;
| [$< lab >$] **return** $< expression >$;
| [$< lab >$] **if** $< expression >< instruction >$
| [$< lab >$] **if** $< expression >< instruction >$ **else** $< instruction >$
| [$< lab >$] **while** $< expression >< instructionBloc >$
| $< instructionBloc >$
$< exprList >$: := $< expression >$
| $< expression >$, $< exprList >$
| $< empty >$
$< lab >$: := $< int >$:

Les différents types d'instruction sont les déclarations de variable locale, les affectations, les appels de méthode, les *return*, les tests, les boucles et les blocs d'instruction.

Une déclaration de variable locale est composée du nom de la variable, de son type et d'une éventuelle affectation. Le choix de considérer une déclaration locale

comme une instruction semble peu intuitif mais ne pose aucun problème au niveau syntaxique.

A chacune des autres instructions peut être associé un label.

Une affectation est composée d'une expression de gauche (toujours un désignateur de variable) et d'une expression de droite.

Un appel de méthode est composé du nom de la méthode auquel est associée une liste d'expressions. De plus un désignateur peut y être ajouté (afin de "situer" la méthode appelée dans le programme). Un test contient une expression (la condition) et une partie "then" qui est elle-même une instruction Il peut contenir ou non une partie "else".

Une boucle est composée d'une expression et d'un bloc d'instructions.

Si les blocs d'instructions sont considérés comme des instructions, c'est une simple astuce permettant une grammaire plus courte. En effet, pour les tests, les instructions du "then" et du "else" peuvent ainsi être soit une simple instruction, soit un bloc.

- **Les expressions**

$$
\begin{aligned}
< \textit{expression} > \quad & ::= \textit{litteral} \\
& \mid < \textit{designateur} > \\
& \mid < \textit{unaryOp} >< \textit{expression} > \\
& \mid < \textit{expression} >< \textit{binaryOp} >< \textit{expression} > \\
& \mid < \textit{expression} > \textbf{instanceOf} < \textit{type} >
\end{aligned}
$$

Une expression peut être un littéral, un désignateur, une opération sur une ou plusieurs expressions et enfin, l'application de la fonction **instanceOf** sur une expression.

- **Les désignateurs**

$$
\begin{aligned}
< \textit{varDesignateur} > \quad & ::= < \textit{varName} > \\
& \mid < \textit{fieldName} > \\
& \mid < \textit{designateur} > . < \textit{fieldName} > \\
< \textit{designateur} > \quad & ::= \textbf{super} \\
& \mid \textbf{this} \\
& \mid \textbf{new} < \textit{className} > (< \textit{exprList} >) \\
& \mid < \textit{varDesignateur} > \\
& \mid < \textit{designateur} > . < \textit{methodName} > (< \textit{exprList} >)
\end{aligned}
$$

Un désignateur peut être un désignateur de variable, un appel de méthode, **this**, **super** et enfin la création d'une nouvelle instance de classe.

Un désignateur de variable est soit un champ, soit une variable locale. Il peut s'agir d'un champ appelé d'une autre classe et donc accompagné d'un désignateur (sensé

désigner cette classe).

- **Les identifiants**

$< className >$ ::= $< id >$
$< methodName >$ ::= $< id >$
$< constructorName >$ ::= $< id >$
$< fieldName >$ ::= $< id >$
$< paramName >$ ::= $< id >$
$< varName >$ ::= $< id >$
$< id >$ ::= **(a|...|z|A|...|Z|_)(a|...|z|A|...|Z|_|0|...|9)***

Tous les identifiants utilisés dans cette syntaxe concrète ont la même syntaxe. Nous ne les différencions que pour des raisons de clarté.

- **Les opérateurs**

$< unaryOp >$ ::= **- | !**
$< binaryOp >$ ::= **&& | || | == | > | < | >= | <= | + | - | % | * | /**

Aucune différenciation n'est faite à ce niveau entre les opérateurs arithmétiques et les opérateurs booléens ou de comparaison.

- **Les littéraux**

$< litteral >$ ::= **null**
| **true**
| **false**
| $< int >$
| $< float >$
| $< String >$
$< int >$ ::= **(0|...|9)(0|...|9)***
$< float >$ ::= **(0|...|9)(0|...|9)*[.(0|...|9)(0|...|9)*][e(0|...|9)(0|...|9)*[.(0|...|9)(0|...|9)*]] (F|f)**
$< String >$ ::= **« (a|...z|A|...|Z|0|...|9|...)* »**

Un littéral peut être un string, un booléen (**true** ou **false**), un entier, un flottant ou enfin la valeur **null**.

La syntaxe abstraite de la figure 1.1 correspond de manière directe à la syntaxe concrète précédente.

1.2 Définition de SAP

SAP (Syntaxe Abstract Point) est une syntaxe abstraite capable de représenter tout programme écrit en VTF simplifié. Il ne s'agit donc pas à proprement parler d'un nouveau

Domaines syntaxiques

$nclass \in \mathbb{N}class$	: Noms de classes
$nfield \in \mathbb{N}field$	: Noms de champ
$nmethod \in \mathbb{N}method$	: Noms de méthode
$nparam \in \mathbb{N}param$	: Noms de paramètre formel
$nvar \in \mathbb{N}var$	: Noms de variable locale
$id \in \mathbb{I}d$	: Identifiants
$lit \in \mathbb{L}it$	: Littéraux
$declfield \in \mathbb{D}eclfield$	: Déclarations de champ
$declmethod \in \mathbb{D}eclmethod$	: Déclarations de méthode
$instr \in \mathbb{I}nstr$	: Instructions
$call \in \mathbb{C}all$	: Appel de méthode
$des \in \mathbb{D}es$	: Désignateurs
$desinst \in \mathbb{D}esinst$	: Désignateurs d'instance
$expr \in \mathbb{E}xpr$	: Expressions
$op \in \mathbb{O}p$	: Opérateurs
$prog \in \mathbb{P}rog$	: Programmes

Syntaxe abstraite

$prog$ ::= $defclass^+$
$defclass$ ::= [**abstract**] $nclass$ [**extends** $nclass$] $declfield^*$ $declmethod^*$ $declconstr^*$
$type$ ::= **null** | **int** | **bool** | **String** | **nclass** | **void**
$declfield$ ::= $type\ nfield$ [= $expr$]
$declmethod$ ::= $type\ nmethod(type\ nparam)^*$
| $type\ nmethod(type\ nparam)^*\ instrbloc$
$declconstr$ ::= $nclass(type\ nparam)^*\ instrbloc$
$instr$ ::= $declvar$
| $instrbloc$
| [lab] **assign** $des\ expr$
| [lab] **if** $expr\ instr$ [$instr$]
| [lab] **while** $expr\ instrbloc$
| [lab] **proc** $desinst\ nmethod\ expr^*$
| [lab] **return** $expr$
$expr$ ::= lit
|$unaryOp\ expr$
|$expr\ binaryOp\ expr$
|$expr$ **instanceOf** $type$
des ::= id | $desinst$. $nfield$
$desinst$ ::= **super** | **this**
| **new** $nclass\ expr$
|des
|$desinst$. $nmethod\ exprList$
lit ::= $desvar$ | $desinst$ | $nvar$ | **null** | **true** | **false** | int | $float$ | $String$

FIG. 1.1 – Syntaxe abstraite de VTF simplifié

langage par rapport à VTF simplifié mais bien d'une transformation de ce langage[2]. L'objectif d'une telle transformation est d'obtenir un langage "proche" du langage initial mais plus simple, plus adapté à des analyses futures. Pour ce faire, ce langage doit être le plus explicite possible (i.e. exprimer les règles de défaut de VTF simplifié) et le plus concis possible. Deux caractéristiques fondamentales de SAP sont d'une part qu'il fonctionne par branchement c'est-à-dire via des points de programme et d'autre part qu'il s'agit d'un langage typé. A chaque expression doit donc être associé un type.

Mentionnons quelques différences par rapport à VTF simplifié :
- la présence d'un label associé à chaque programme et désignant la première instruction de la méthode principale[3] ;
- la présence obligatoire d'au moins un constructeur explicite par classe ;
- l'impossibilité d'initialiser un champ ou une variable locale ;
- la disparition de la notion de bloc au sein d'une méthode et par conséquence, l'obligation de nommer différemment les variables locales d'une même méthode ;
- l'obligation de passer par une variable intermédiaire lors des appels de fonction utilisés à l'intérieur d'une expression quelconque et ce, afin d'éviter les effets de bord lors de l'évaluation d'une expression ;
- l'obligation de terminer chaque méthode par une instruction `return` ;
- la présence de labels en début d'instruction et éventuellement à la fin pour identifié l'instruction suivante,
- l'absence des boucles et des tests qui doivent être traduits en utilisant les branchements.

1.3 De VTF simplifié à SAP : exemple

Nous prenons ici un exemple de programme écrit successivement en VTF-simplifié et en SAP. Ce programme ne fait pas grand chose d'intéressant mais a le mérite d'illustrer les principales différences entre les deux langages. Les commentaires sont bien sûr absents en SAP mais sont ici insérés afin de fournir quelques éclaircissements.

1. Le programme en VTF simplifié :

```
class Operation{
```

[2]En ce sens, parler de *langage SAP* est un abus de langage. En effet, il semble plus juste de considérer SAP comme une *syntaxe abstraite* particulière de VTF simplifié. Il est cependant plus simple et intuitif de parler d'une traduction d'un *langage* à un autre plutôt que d'une *syntaxe* à une autre.

[3]Notons qu'en VTF, si le parser ne s'en soucie guère, l'absence de méthode principale poserait évidemment des problèmes lors d'une éventuelle exécution du programme VTF. Dès lors, nous imposons tacitement qu'un programme VTF simplifié aie une méthode **main()**. Dans les deux langages, cette méthode est syntaxiquement identique à toute autre méthode mais a sémantiquement le rôle particulier de "première méthode exécutée".

$$
\begin{array}{lll}
prog & ::= & defclass^{+}\, lab \\
defclass & ::= & [\textbf{abstract}]\ nclass\ [\textbf{extends}\ nclass]\ declfield^{*}\, declmethod^{*}\, declconstr^{+} \\
type & ::= & \textbf{null} \mid \textbf{int} \mid \textbf{bool} \mid \textbf{String} \mid \textbf{nclass} \mid \textbf{void} \\
declfield & ::= & type\ nfield \\
declmethod & ::= & type\ nmethod\ (type\ nvar)^{*} \\
 & & \mid type\ nmethod\ (type\ nvar)^{*}\ (type\ nvar)^{*}\ lab\ instr^{+} \\
declconstr & ::= & nclass(type\ nvar)^{*}(type\ nvar)^{*} labinstr^{+} \\
 & & \mid nclass(type\ nparam)^{*}(type\ nvar)^{*} labinstr^{*}\ lab\ \textbf{super}\ expr^{*}\ lab\ instr^{+} \\
 & & \mid nclass(type\ nparam)^{*}(type\ nvar)^{*} labinstr^{*}\ lab\ \textbf{this}\ expr^{*}\ lab\ instr^{+} \\
instr & ::= & lab\ \textbf{assign}\ des\ expr\ lab \\
 & & \mid lab\ \textbf{if}\ lab\ lab \\
 & & \mid lab\ \textbf{skip}\ lab \\
 & & \mid lab\ \textbf{proc}\ call\ lab \\
 & & \mid lab\ \textbf{return}[expr] \\
 & & \mid lab\ \textbf{fonc}\ nvar\ call\ lab \\
 & & \mid lab\ \textbf{constr}\ nvar\ nclass\ expr^{*}\ lab \\
call & ::= & type\ desinst\ nmethod\ expr^{*} \\
 & & \mid type\ \textbf{super}\ nmethod\ expr^{*} \\
des & ::= & type\ nvar \mid type\ \textbf{super}\ nfield \\
 & & \mid type\ desinst\ nfield \\
desinst & ::= & type\ \textbf{this} \mid des \\
expr & ::= & \textbf{null null} \mid type\ litt \mid type\ op\ expr^{*} \\
 & & \mid desinst \mid \textbf{bool}\ expr\ \textbf{instanceof}\ type
\end{array}
$$

FIG. 1.2 – Syntaxe abstraite de SAP

```
    void main(){
    // Nous appelons cette méthode "main" afin que le traducteur
    // puisse lui faire correspondre le label du programme SAP.
        ExprLeft exprLeft=new ExprLeft(2F);
        ExprRight exprRight=new ExprRight(3);
        boolean b=true;
        while(b){
            b=exprLeft.operation(5)!=exprRight.operation(7))
        }
    }
}
class ExprLeft{
    int _x;
    ExprLeft(float x){_x=(int)x;}
    int operation(int z){return _x*z;}
}
class ExprRight{
    int _x;
    ExprRight(int x){_x=x;}
    int operation(int z){return _x+z;}
}
```

2. le programme en SAP :

```
class Operation{
    Operation(){
    // Un constructeur explicite est ajouté à cette classe.
        begin at lab 0;
        // L'instruction où commence le constructeur doit être
        // annoncée.
        0 return;
        // Cette instruction est ajoutée parce que tout constructeur
        // ou méthode doit se terminer par un return.
    }
    void main(){
        ExprLeft exprLeft;
        ExprRight exprRight;
        boolean b;
        // Une variable ne peut pas être affectée dès sa déclaration.
        // L'affectation se fait donc après (point de programme 2).
        int v4;
        int v5;
        // Ces variables intermédiaire ont du être ajoutée
```

```
        begin at lab 0;
        0 exprLeft=new ExprLeft(2F) 1;
        1 exprRight=new ExprRight(3) 2;
        2 b=true 3;
        3 if b 4 7;
        // La boucle est transformée en test.
        4 v4=exprLeft.operation(5) 5;
        5 v5=exprRight.operation(7) 6;
        6 b=v4!=v5 3;
        // Une expression ne peut être composée d'appels de méthodes.
        // Des variables intermédiaires sont donc insérées
        // dans le programme.
        7 return;
    }
class ExprLeft{
    int _x;
    ExprLeft(float x){
        begin at 0;
        0 _x=(int)x 1;
        1 return;
    }
    int operation(int z){
        begin at 0;
        0 return _x*z;
    }
}
class ExprRight{
    int _x;
    ExprRight(int x){
        begin at 0;
        0 _x=x 1;
        1 return;
    }
    int operation(int z){
        begin at 0;
        0 return _x+z;
    }
}
label de début du programme Operation_main_0
```

Chapitre 2

Traduction : explications générales

Pour réaliser la traduction de VTF simplifié à SAP, il existe sans aucun doute plusieurs façons de faire. Nous expliquons ici le schéma général suivant lequel nous avons décidé de travailler. Le détail des différentes étapes fera l'objet des chapitres suivants.

Nous rappelons égalemnt dans ce chapitre quelques concepts théoriques nécessaires à notre démarche de travail. Ces rappels sommaires favorisent l'intuition. Pour plus de rigueur et de détails, le lecteur se référera à [Sch00, WM94].

2.1 Schéma général de la traduction

Nous avons choisi une traduction d'un programme de VTF à SAP se réalisant en plusieurs phases.

Tout d'abord, il s'agit de *parser* le programme, c'est-à-dire de créer un arbre abstrait représentant le programme par la syntaxe abstraite du langage initial.

Cette analyse se réalise en une fois. En d'autres mots, le parser reçoit le flux de caractères représentant le programme et le transforme en un arbre en une seule étape. Cependant, conceptuellement, on peut diviser le parser en trois sous-étapes : l'analyse lexicale qui à partir d'un flux de caractères renvoie un flux de mots, le filtrage qui de ce flux de mots extrait ceux qui doivent être traités par la suite et enfin l'analyseur syntaxique qui crée l'arbre abstrait.

La phase suivante est le *type checking*. Celui-ci se réalise sur l'arbre créé par le parser. Il s'agit de vérifier l'aspect "bien formé" du programme (par exemple, qu'il n'y a pas deux classes de même nom) ainsi que la compatibilité entre les types des expressions. Il faut, par exemple, que le type de l'expression de droite d'une affectation soit une spécialisation

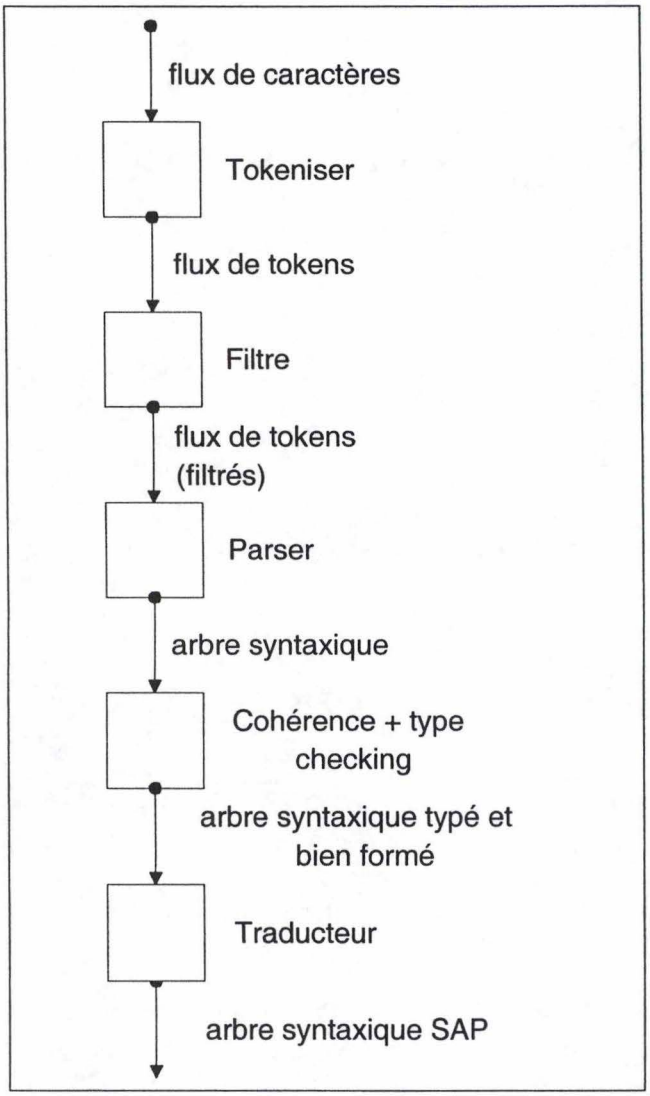


FIG. 2.1 – Les différentes étapes du traducteur

du type de l'expression de gauche.

Une fois cet arbre abstrait certifié bien formé et bien typé, nous pouvons le traduire en un nouvel arbre abstrait correspondant cette fois à la syntaxe abstraite SAP.

Ces différentes phases sont reprises dans la figure 2.1.

2.2 Quelques rappels

Le rôle d'un *parser* est de traduire un programme écrit dans un certain langage en la syntaxe abstraite de ce langage. Usuellement, un *parser* se réalise en trois étapes : l'analyse lexicale, le filtrage et enfin l'analyse syntaxique.

2.2.1 Analyse lexicale

L'analyse lexicale d'un programme est effectuée par l'analyseur lexical (aussi appelé *tokeniser*). La mission de celui-ci est de reconnaître les *mots* (*token*) du programme. Un *mot* est une suite de caractères appartenant à la grammaire du langage.

Pour réaliser un analyseur lexical, il s'agit tout d'abord d'exprimer les différents mots du langage en *expressions régulières* (cf. Définition 1).

Définition 1 *Une expression régulière est une expression écrite avec*

$$a, \emptyset, \epsilon, ., \cup, *$$

(. est souvent omis et $\cup$ s'écrit aussi $|$ ou $+$)

La deuxième opération consiste à exprimer chaque expression régulière en un *automate fini* (cf Définition 2) grâce au théorème 1.

Définition 2 *Un automate fini est un 5-tuple $(\Sigma, Q, q_0, F, \Delta)$ où :*

1. *Σ est un ensemble fini (appelé alphabet) ;*
2. *Q est l'ensemble fini d'états ;*
3. *$q_0 \in Q$, est l'état initial ;*
4. *$F \subseteq Q$, est l'ensemble des états finaux ;*
5. *$\Delta \subseteq Q \times (\Sigma \cup \{\epsilon\}) \times Q$ est la relation de transition.*

Définition 3 *Le langage d'un automate est l'ensemble des mots acceptés par cet automate. Un mot est accepté par un automate si on peut trouver un chemin c*
- *qui commence dans l'état initial,*
- *qui finit dans un état final,*
- *qui suit des transitions,*
- *tel que la concaténation des étiquettes des transitions du chemin c donne m.*

Théorème 1 Théorème de Kleene
Un langage est régulier ssi c'est le langage d'un automate fini (cf. Définition 3)

Nous transformons ensuite cet automate fini en un automate *déterministe*[1] et *complet* (Définition 4).

Définition 4 *Un automate fini est déterministe si sa relation de transition est de la forme* $\Delta \subseteq (Q \times (\Sigma)) \rightarrow Q$.

Intuitivement, un automate est déterministe s'il ne peut se comporter que d'une façon pour un mot donné, c'est-à-dire qu'il n'a pas de transition-ϵ et, qu'étant donné un état et un caractère lu, il y a au plus 1 état d'arrivée.

Dans ces automates, chaque transition représente la lecture d'un caractère. Arriver à un état final signifie la fin de la lecture d'un token valide.

Pour finir, nous devons construire l'union de tous les automates représentant un token. Nous obtenons ainsi un grand automate fini déterministe[2] capable d'identifier tous les types de token.

L'analyseur lexical doit être capable de créer, à partir d'un flux de caractères et de cet automate fini déterministe général, un flux de token. Pour cela, il parcourt le flux de caractères et crée un token chaque fois que l'état courant est un état final et que, de cet état, il n'existe aucune transition possible vers un autre état final. Cette deuxième condition s'explique par le fait que nous recherchons systématiquement le plus grand token valide.

Par exemple, face au flux de caractères `23f`, si on s'arrêtait au premier état final, on obtiendrait que ce flux représente la suite $< entier >< entier >< identifiant >$ alors qu'il s'agit naturellement d'un <flottant>.

[1] Cette transformation se fait en suivant la théorie dite de la *détermination*. Dans le cas de VTF simplifié, obtenir que les automates soient déterministes est cependant assez intuitif et nous ne nous étendons donc pas sur cette théorie.

[2] A nouveau, la théorie de la détermination est impliquée de façon "implicite"

2.2.2 Filtrage

Le filtrage permet à partir d'un flux de token d'éliminer les token dénués d'influence dans la suite du parser. Par exemple les blancs ou les commentaires n'interviennent ni dans l'analyse syntaxique ni dans le typechecking ni dans aucune autre analyse ultérieure. Ils sont donc inutiles.

2.2.3 Analyse syntaxique

Après l'analyse lexicale, on peut procéder à l'analyse syntaxique. Cette analyse est réalisée par l'analyseur syntaxique aussi appelé parser[3] Celui-ci peut être considéré comme une boîte qui, à partir d'un flux de token (envoyé par l'analyseur lexical), construit un arbre représentant le programme traduit dans la syntaxe abstraite du langage. Cet arbre est appelé *arbre abstrait.* Il s'agit donc ici de vérifier la validité syntaxique du programme et dans le même temps de supprimer le "sucre syntaxique" du programme.

Pour réaliser le parser, on définit pour commencer le langage sous la forme de sa grammaire BNF. Cette grammaire est dite *non contextuelle* (cf. Définition 5).

Définition 5 *Une grammaire non contextuelle contient 4 éléments :*
- *l'ensemble des terminaux* V_R,
- *l'ensemble des non-terminaux* V_N,
- *l'ensemble des productions* P,
- *le symbole de départ* $s \in V_N$.

où une production se compose de :
- *un non-terminal appelé partie gauche de la production,*
- *une suite de terminaux et non-terminaux appelée partie droite de la production*

Un exemple de production est :

$$< instruction > ::= \textbf{if } < expr > < instr > \textbf{ else } < instr > \in P.$$

A partir de cette grammaire, on peut choisir de faire une analyse *descendante* ou *ascendante* du programme. Une analyse descendante se caractérise par le fait que l'on doit pouvoir choisir la production correspondant au flux de token sur base d'un nombre fini et déterminé de symboles à lire. L'idée de l'analyse syntaxique ascendante est de choisir la production à appliquer après avoir lu le texte correspondant.

Nous avons choisi l'analyse descendante pour des raisons de simplicité. Une analyse descendante doit se faire à partir d'un grammaire LL(k), c'est-à-dire une grammaire

[3]Le terme *parser* est ici ambigu. En effet il est utilisé à la fois pour la succession des trois étapes et parfois pour la simple analyse syntaxique.

telle que le choix de la bonne expansion doit pouvoir se faire sur base des k prochains symboles à lire. Notons que ce qui correspond à une production dans une grammaire non-contextuelle est appelé une *règle* dans la grammaire LL(k).

Nous avons choisi, à nouveau pour des raisons de simplicité, une grammaire LL(1). En effet, appliquer une analyse descendante sur une grammaire LL(1) ne provoque jamais d'indétermination puisque, dès le prochain token, on sait quelle règle il faut appliquer. Cette analyse ne nécessite donc pas de retour en arrière dans la grammaire et nous évitons ainsi tous les soucis de sauvegarde de token que cela pourrait engendrer.

Il s'agit donc de transformer la grammaire BNF en une grammaire LL(1). Cela se fait notamment en *supprimant la récursivité à gauche* et en *factorisant les parties communes.*

Eliminer la récursivité à gauche, c'est, si A est un non terminal, transformer $A ::= A\alpha \mid \beta$ en $A ::= \beta A'$, $A' ::= \alpha A' \mid \epsilon$.

Factoriser le parties communes consiste à postposer le choix en regroupant les parties communes de deux règles. Par exemple,

$$I ::= \textbf{if } E \textbf{ then } I \textbf{ else } I$$

devient

$$I ::= \textbf{if } E \textbf{ then } I \textbf{ else } C$$
$$C ::= \epsilon \mid \textbf{else } I$$

La grammaire LL(1) obtenue, il s'agit de créer les liens entre les différentes règles. Pour cela, nous associons à chaque règle un ensemble de prédictions *predictset.* Associer un predictset à une règle doit permettre de savoir quels terminaux (c'est-à-dire quels token) peuvent être lus pour que cette règle puisse être appliquée.

Une fois les predictset définis, nous insérons aux endroits adéquats de la grammaire LL(1) des *actions.* Chacune de ces actions crée un bout de l'arbre abstrait à partir de ses fils qui avaient été préalablement obtenus. Ainsi, les actions sont réalisées au fur et à mesure que le flux de token est parcouru et, finalement, nous obtenons l'arbre abstrait correspondant au programme dans son entièreté.

Chapitre 3

Parser

L'objectif du parser est la vérification syntaxique d'un programme. Il prend donc comme argument un programme et fournit comme résultat, soit un arbre abstrait représentant le programme par la syntaxe abstraite du langage, soit une erreur (en cas de faute syntaxique).

Dans ce chapitre, nous décrivons les trois étapes successives nécessaires à la conception d'un parser : L'analyse lexicale, le filtrage et l'analyse syntaxique. De ces étapes sont expliquées la conception et l'implémentation.

3.1 Analyse lexicale

Première étape dans la conception du traducteur, l'analyseur lexical ou *tokeniser* doit reconnaître les mots (*token*) du programme. Le résultat obtenu est soit un flux de token, soit une erreur si le tokeniser tombe sur un symbole inexistant dans le langage ou sur une suite de symboles invalides. De plus, notre tokeniser associe à chaque token sa position dans le fichier d'entrée. Cette position est composée du numéro de ligne et du numéro de colonne du premier symbole du token.

3.1.1 Conception

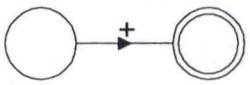
Pour réaliser le tokeniser, nous recensons tout d'abord l'ensemble des token existants dans le langage.

Nous exprimons ensuite chaque token sous la forme d'une expression régulière. Dans la syntaxe concrète de VTF simplifié (section 1.1.1), les token entier ($< int >$), flottant ($<$

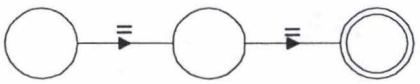
float >), identifiant (< *id* >) et string (< *String* >) sont déjà définis comme expressions régulières. Il est trivial de transformer les autres token en expressions régulières.

A partir de ces expressions régulières, nous créons un automate fini déterministe et complet pour chaque token. Ci-dessous, à titre d'exemple, se trouvent les automates pour l'affectation, l'égalité, l'addition, les entiers, les flottants et les identifiants.

1. l'addition :

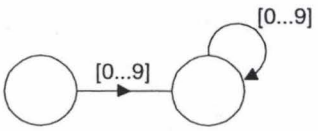


2. l'égalité :

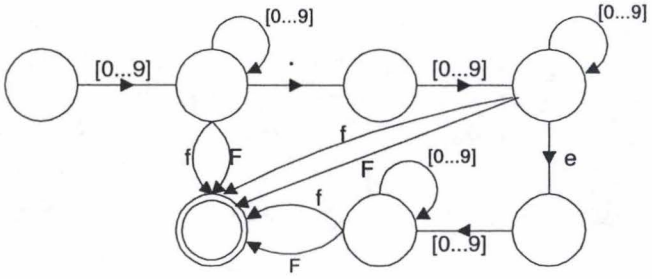


3. l'affectation :

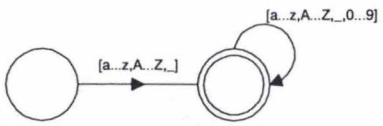
4. les entiers :



5. les flottants :

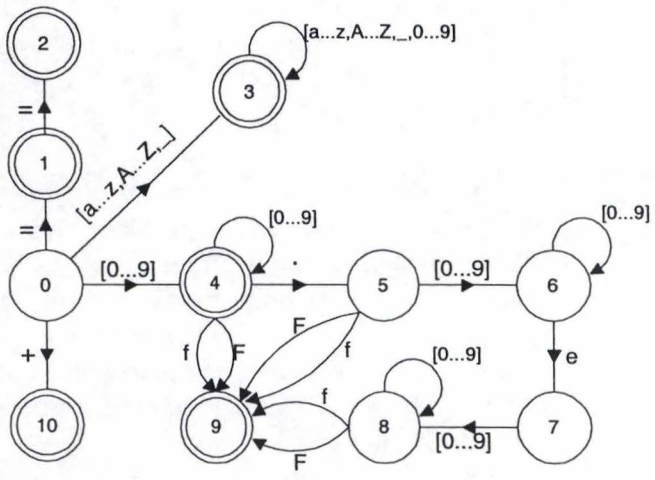


6. les identifiants :



Nous pouvons alors rassembler tous ces automates pour en créer un seul pour tous les token. C'est cette automate final qui est parcouru par l'analyseur lexical et qui représente en fait la boîte transformant un flux de caractères en un flux de token (i.e le tokeniser).

Si on reprend par exemple les automates ci-dessus, nous pouvons obtenir en les mettant ensemble l'automate suivant :



avec
- l'état 0 est l'état initial ;
- l'état final 1 correspond au token = (affectation) ;
- l'état final 2 correspond au token == (égalité) ;
- l'état final 3 correspond à un token identifiant ;
- l'état final 4 correspond à un entier ;
- l'état final 9 correspond à un flottant ;
- l'état final 10 correspond à l'addition.

3.1.2 Implémentation

Le tokeniser est une boîte qui reçoit un flux de caractères (i.e un fichier) et renvoit les token un à un.

Notre tokeniser est implémenté à l'intérieur du package `tokeniser`. Dans ce package, chaque token est représenté par une classe qui hérite de la classe générique `Token`. Nous avons cependant rassemblé certains token afin qu'ils soient définis par la même classe. Il est par exemple inconcevable de représenter chaque identifiant existant dans le langage par une classe différente car cela engendrerait un nombre quasi infini de classes. Les token sont donc rassemblés par type suivant le tableau 1.

Types de token	Descriptions
les littéraux[1]	les string, les entiers, les flottants et les identifiants sont des token particuliers : leur type ne suffit pas à les définir. Ces token sont identifiés par leur type et par leur valeur c'est-à-dire la suite de symboles qui les composent. Ils sont définis par la même classe, une instance de celle-ci étant paramétrée par la valeur de l'identifiant qu'elle représente.
les mots clés	il s'agit d'identifiant qui, par leur valeur pariculière, ont une signification bien spécifique. Ils ont donc un statut particulier. Contrairement aux identifiants, la suite de caractères composant le mot clé en détermine son type et non sa valeur. Cependant, ils sont tous définis par la même classe, une instance de celle-ci étant paramétrée par un numéro identifiant le mot-clé lui correspondant
les autres	=, ==, {, ... Chacun de ces token est représenté par une classe distincte.

TABLEAU 1 : Les différents types de token

La classe `Tokeniser` est la classe où sont définis l'automate fini déterministe exprimant l'ensemble des token et la méthode parcourant cet automate et renvoyant les token successifs. Dans cette classe sont aussi mis à jour continuellement les numéros de ligne et de colonne où se trouve le dernier caractère lu dans le fichier.

Trois objets servent à représenter l'automate. Le premier de ces objets est la matrice `_matrixForStates[][]` qui représente toutes les transitions possibles. Elle a en ordonnée les états de l'automate et en abscisse tous les caractères susceptibles d'être lus. Elle désigne ainsi selon l'état courant et le dernier caractère lu l'état suivant. Par convention, si de l'état courant et du caractère lu, aucune transition n'est possible, l'état suivant redevient l'état 0 c'est-à-dire l'état initial.

A cette matrice est associée une autre matrice, de booléens cette fois, `_matrixForSave[][]` qui pour chaque transition, détermine si le dernier caractère lu doit être sauvé ou pas. S'il s'agit d'une transition susceptible de créer un token identifiant par exemple, il faut sauver le dernier caractère car il fait partie de la valeur de l'identifiant.

A cela est ajouté un tableau de booléens `_finalState[]` qui détermine pour chaque état s'il est final ou pas.

Dans la classe `Tokeniser` est également définie une méthode `createToken(int state)` qui, à partir de l'état final où l'on se trouve, crée le token correspondant. S'il s'agit d'un token de type identifiant, mot clé, string, entier ou flottant, un argument en plus reprenant la valeur du token doit être considéré.

Enfin, la méthode `getToken(Filereader file)` est la méthode principale du tokeniser qui renvoie un à un les token (i.e au parser qui en fera bon usage). Cette méthode lit le fichier caractère par caractère et, à chaque caractère lu, agit comme suit :

- si l'état n'est pas final et qu'une transition est possible : passer à l'état et au caractère suivant ;
- si l'état est final et qu'une transition est possible : passer à l'état et au caractère suivant ;
- si l'état n'est pas final, qu'aucune transition n'est possible et que l'on n'est passé par aucun état final depuis que le token précédant a été envoyé : générer une exception de type `LexicalException` et arrêter l'analyse lexical ;
- si l'état n'est pas final, qu'aucune transition n'est possible mais que l'on est déjà passé par un état final depuis que le token précédant a été envoyé : créer le token correspondant à ce dernier état final (via la méthode `createToken(int state)`) et recommencer le traitement à partir du caractère suivant ce dernier token et de l'état 0.
- si l'état est final et qu'aucune transition n'est possible, renvoyer le token correspondant à ce dernier état final, passer au caractère suivant et réinitialiser l'état courant.

3.2 Filtrage

Le filtre peut être défini comme une boîte qui, recevant un token, détermine si celui-ci est nécessaire pour la suite des opérations c'est-à-dire pour l'analyse syntaxique. Dans le cas présent, les mots clés **public**, **protected** et **private** sont inutiles car, s'ils sont acceptés par le tokeniser (et ce dans le cadre d'éventuelles extensions futures du langage) ils ne seront pas traités syntaxiquement. De même, les commentaires et les espaces ne passeront pas le cap du filtre car il est évidemment inutile de les traiter syntaxiquement.

Le filtrage se fait via la méthode $getToken(FileReader file)$ se trouvant dans la classe *Filter*. Elle se contente d'appeler la méthode $getToken(FileReader file)$ de la classe *Tokeniser*. Ensuite, selon l'utilité du token, soit elle renvoie le token, soit elle passe directement au token suivant.

3.3 Analyse syntaxique

Le parser consiste en une boîte qui, recevant en entrée un flux de token, forme progressivement un arbre abstrait qui correspond au programme exprimé à partir de la syntaxe abstraite du langage VTF simplifié. Bien sûr, s'il existe une erreur syntaxique dans le programme, le parser interrompt la compilation et se charge de signaler l'erreur à l'utilisateur.

3.3.1 Conception

Pour réaliser le parser du langage VTF simplifié, nous sommes partis de sa syntaxe concrète exprimée en BNF (voir section 1.1.1). Comme notre décision est de faire l'analyse descendante du programme nous avons ensuite transformé cette grammaire BNF en grammaire LL(1). Ensuite, nous avons associé à chaque règle un ensemble de prédictions et nous avons inséré dans les règles les actions à réaliser. A partir de là, nous pouvons maintenant appliquer les règles au flux de token afin de créer l'arbre abstrait.

Dans la suite de cette section, nous passons en revue chacune de ces étapes à partir d'un petit morceau de la grammaire BNF du langage reprenant uniquement :
- les opérations binaires d'addition, de soustraction, de multiplication, de division et de modulo ;
- l'opération unaire d'inverse ;
- les entiers.

$$
\begin{aligned}
&< \textit{expression} > && ::= \textit{litteral} \\
&&& \quad | \text{ - } < \textit{expression} > \\
&&& \quad | < \textit{expression} > < \textit{binaryOp} > < \textit{expression} > \\
&< \textit{binaryOp} > && ::= + \mid \text{-} \mid * \mid / \mid \% \\
&< \textit{litteral} > && ::= < \textit{int} > \\
&< \textit{int} > && ::= (\mathbf{0}|...|\mathbf{9})(\mathbf{0}|...|\mathbf{9})^*
\end{aligned}
$$

On constate aisément que si on devait appliquer ces règles, nous tomberions vite sur une indétermination. Par exemple, face à la simple expression 1+4, en lisant le premier token (un entier) on ne peut pas s'il faut appliquer la règle $< \textit{expression} > ::= \textit{litteral}$ ou la règle $< \textit{expression} > ::= < \textit{expression} > < \textit{binaryOp} > < \textit{expression} >$.

C'est pourquoi nous devons transformer cette grammaire en grammaire LL(1). Ensuite, nous devons associer à chaque règle un prédictset et insérer les actions. C'est ici que, pour le morceau de grammaire ci-dessus, nous obtenons la grammaire LL(1) de la figure 3.1 :

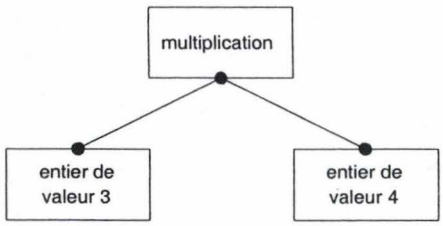
AC1,...,AC6 sont les actions. C'est la succession de ces actions qui permet de créer l'arbre abstrait. Nous pouvons représenter ces actions par des bouts de code. Par exemple, l'action AC3 est repésentée comme suit :

```
ASTExpression right = (ASTExpression)ps.pop();
ASTExpression left = (ASTExpression)ps.pop();
ps.push(new ASTMultiplication(left, right));
```

Ainsi, grâce à cette action, la lecture du flux 3*4 provoquerait la création du sous-arbre abstrait suivant :

Règles		**Predictsets**
$< expression >$	$::= < term >$	
$< term >$	$::= < factor > < termSfx >$	
$< termSfx >$	$::= + < factor >$ AC6 $< termSfx >$	{+}
	$\mid$ - $< factor >$ AC5 $< termSfx >$	{-}
	$\mid < empty >$	
$< factor >$	$::= < litteral > < factorSfx >$	{-,entier}
$< factorSfx >$	$::= \% < litteral >$ AC4 $< factorSfx >$	{%}
	$\mid$ * $< litteral >$ AC3 $< factorSfx >$	{*}
	$\mid$ / $< litteral >$ AC2 $< factorSfx >$	{/}
	$\mid < empty >$	{+,-}
$< litteral >$	$::=$ - $< litteral >$ AC1	{-}
	$\mid$ entier	{entier}

FIG. 3.1 – Grammaire LL(1) d'un sous-ensemble de VTF simplifié



Remarquons que le fait d'avoir considéré VTF simplifié comme un langage de type LL(1) et donc déterministe nous a imposé d'être particulièrement permissifs dans certaines règles de la grammaire LL(1). L'exemple le plus frappant est sans doute l'autorisation de "caster" une expression par une autre expression et non nécessairement par un type ! Ainsi de tels erreurs syntaxiques ne seront corrigées qu'au niveau de l'étape suivante c'est-à-dire au niveau des vérifications de la cohérence du programme.

3.3.2 Implémentation

Notre parser reçoit un à un les token du tokeniser (plus exactement du filtre). Chaque token lui permet d'avancer dans la grammaire LL(1) et de soit créer l'arbre abstrait, soit s'interrompre pour signaler une erreur syntaxique.

Le parser est implémenté à l'intérieur du package `parser`. Dans ce package se trouve notamment la classe `Parser` dans laquelle sont définies les méthodes qui représentent les règles de la grammaire LL(1). Dans cette classe se trouve également la pile `_ps` permettant d'accomplir les actions.

La création d'un sous-arbre par une action correspond à la création d'une instance de la classe représentant ce sous-arbre. Cette instance de classe reprend donc un pointeur vers chacun de ses fils. C'est ainsi que par exemple, l'action

```
ASTExpression right = (ASTExpression)_ps.pop();
ASTExpression left = (ASTExpression)_ps.pop();
_ps.push(new ASTMultiplication(left, right));
```

crée un instance de la classe `ASTMultiplication` implémentée comme suit :

```
public class ASTMultiplication extends ASTFactor{
    private ASTFactor _operand1;
    private ASTLiteral _operand2;
    public ASTMultiplication(ASTFactor op1, ASTLiteral op2) {
        _operand1=op1;
        _operand2=op2;
    }
}
```

où la classe `ASTFactor` étend la classe `ASTExpression` et la classe `ASTLiteral` étend la classe `ASTFactor`.

Une fois tous ces éléments implémentés, le parser fonctionne comme suit.

1. Il reçoit en entrée un fichier et appelle la méthode `getToken(FileReader file)` du filtre qui lui renvoie le premier token (utile) du fichier.
2. Il applique alors la première règle de la grammaire LL(1) (à savoir, $program$ ::= $declass\ programSfx$) qui est représentée par une méthode (`program`) qui se trouve dans la classe `parser`.
3. A présent lancé, le parser applique les règles successives. Dans la pile `_ps` se trouve des sous-arbres de plus en plus grands qui, à la fin seront rassemblé pour former l'arbre abstrait final. Cependant, si le parser tombe sur un token qui ne satisfait aucune règle, une exception de type `ParseException` est créée et une erreur syntaxique est renvoyée au programmeur avec l'endroit où est l'erreur et une piste vers la solution (la liste des token qui auraient été valables).

Chapitre 4

Le type checking

Nous entendons par *type checking* la vérification des caractères "bien formé" et "bien typé" d'un programme.

Vérifier qu'un programme est bien formé consiste, par exemple, à s'assurer que deux classes de ce programme n'ont pas le même nom. L'importance de ce travail de vérification dépend de la permissivité du parser[1]. On peut, par exemple, imaginer un parser qui autorise de mettre des méthodes abstraites dans une classe concrète et reporte l'erreur au type checking. Un autre parser pourrait par contre provoquer une erreur immédiatement (c'est le cas du nôtre).

La vérification des types a pour but de contrôler la compatibilité entre les types des expressions. Il faut, par exemple, que le type de l'expression de droite d'une affectation soit une spécialisation du type de l'expression de gauche. Notre "type checker" ne se contente pas d'une simple vérification au sens propre mais garnit également le programme des annotations de type et ce, en vue de la traduction en SAP (voir 1.2 Définition de SAP).

Cette phase de type checking commence donc lorsque le programme est parsé et prend comme argument l'arbre abstrait créé par le parser. Le résultat est un arbre abstrait *typé*.

Dans ce chapitre, nous décrivons la conception puis l'implémentation de notre type checking.

Au niveau de la conception, notons que nous favorisons à nouveau l'aspect intuitif par rapport à l'aspect formel. Nous introduisons les différentes notions utilisées et les appliquons à notre langage. Ensuite, nous expliquons de manière informelle les règles de type checking à respecter.

---

[1] On peut d'ailleurs considérer que cette phase du travail n'a pas sa place dans le chapitre *type checking* mais l'aurait plutôt dans le chapitre *analyse syntaxique*. Cependant, nous en parlons dans ce chapitre-ci car sa réalisation se fait dans le même temps que la vérification des types.

4.1 Conception

Pour concevoir notre type checking, nous commençons par recenser l'ensemble des vérifications à effectuer. Il y a les vérifications liées au caractère bien formé des programmes et les vérifications liées à leur caractère bien typé. Pour réaliser ces vérifications, nous définissons les concepts de *portée* et de *visibilité* d'une déclaration.

La portée est la zone textuelle du programme dans laquelle une déclaration est valable. Par exemple, en VTF simplifié, la portée d'une variable locale est le bloc où elle est déclarée tandis que la portée d'une méthode est la classe où elle est déclarée ainsi que les classes qui l'étendent.

Les déclarations peuvent cependant, à l'intérieur même de leur portée, être cachées par d'autres déclarations. Par exemple, dans le programme

```
class A{
    foo(int i){...}
}
class B extends A{
    foo(int i){...}
}
```

La portée de la méthode `foo(int i)` de la classe `A` est `A` et `B`. Cependant, la méthode `foo` de la classe `A` n'est plus accessible dans `B` puisque celle-ci est *cachée* par la méthode `foo` de la classe `B`. On dit que la *visibilité* de la méthode `foo` de `A` est `A`.

Nous reprenons ci-après les différentes portées et visibilités à déterminer dans le cadre de notre langage.

- **les déclarations de variables locales**
  - portée : le bloc où se trouve la déclaration et ses sous-blocs
  - visibilité : idem
- **les déclarations de paramètres**
  - portée : la méthode où est déclaré le paramètre
  - visibilité : idem
- **les déclarations de champs**
  - portée : la classe où est déclaré le champ et les classes spécifiant cette classe (la classe qui l'étend, la classe qui étend la classe qui l'étend,...).
  - visibilité : la classe où est déclaré le champ et les classes spécifiant cette classe jusqu'à ce qu'éventuellement un champ du même nom soit déclaré.

- **les déclarations de méthodes**
  - portée : la classe où est déclarée la méthode et les classes spécifiant cette classe.
  - visibilité : idem jusqu'à ce qu'éventuellement, on ne tombe sur une méthode de même signature
- **les déclarations de constructeurs**
  - portée : toutes les classes du programme
  - visibilité : toutes les classes du programme

Il y a conflit entre deux déclarations de variable si elles ont le même nom (pour les variables) ou la même signature (pour les méthodes et constructeurs)[2] et qu'il existe une zone textuelle appartenant à la visibilité des deux.

Une fois les portées et visibilités des déclarations déterminées, nous pouvons typer chaque identificateur sur lequel nous tombons. Par exemple, si l'expression `x+2` se trouve dans la visibilité de la déclaration `int x`, alors le type de `x` est **int**. Nous pouvons également, de façon triviale, typer les littéraux (par exemple, le type de 2 est **int**).

Lorsque les littéraux et les identifiants sont typés, nous pouvons typer les expressions en utilisant certaines règles (cf. 4.1.2 Règles de typage). Si l'on reprend l'expression trivial `x+2`, le type checking donne comme résultat l'expression `(x[int]+2[int])[int])`. S'il y a une erreur de type, le type checking n'est pas interrompu, mais l'erreur est propagée. Par exemple, le typage de l'expression `((x+2)*4)` se trouvant cette fois dans la visibilité d'une déclaration `A x` où `A` est une classe du programme, donnera

$$((\texttt{x[A]} + 2\texttt{[int]})\texttt{[error]} * 4\texttt{[int]})\texttt{[error]}.$$

Nous déterminons ci-après les vérifications essentielles à considérer pour s'assurer du caractère bien formé du programme. Ensuite, nous montrons via quelques exemples en quoi consiste la vérification des types.

Enfin, nous expliquons comment nous avons conçu notre type checking étant donné les vérifications à réaliser.

4.1.1 Vérifications du caractère bien formé

Ci-dessous, nous reprenons les vérifications essentielles qui doivent être effectuées sur l'arbre abstrait afin de s'assurer du caractère bien formé du programme.

1. vérifier qu'il n'existe pas deux classes de même nom.

[2] La signature d'une méthode est son nom, le nombre de paramètres et le type des paramètres

2. vérifier qu'il n'existe pas de cycle au niveau de l'héritage des classes (ex : A extends B, B extends C et C extends A).
3. vérifier que toutes les méthodes déclarées dans une classe abstraite sont bien implémentées dans les classes concrètes l'étendant.
4. vérifier qu'il n'existe pas deux méthodes de même signature (nom de la méthode, nombre de paramètres et types des paramètres) à l'intérieur d'une même classe.
5. vérifier que, si différentes méthodes appartenant à différentes classes parentes entre elles ont la même signature, elles renvoient bien le même type.
6. vérifier l'absence de conflit entre les déclarations de champs, de paramètres, de variables locales. Par exemple, il ne peut pas y avoir deux champs de même nom dans la même classe ou deux paramètres de même nom dans la même signature de méthode.
7. vérifier toutes les expressions castées le sont bien par des types et non par des expressions[3].

4.1.2 Règles de typage

Une sémantique formelle exprimant des règles de typage est spécifiée dans [Pol99][4]. Dans ce mémoire, il s'agit des règles de typage de SAP mais celles-ci sont fort proches des règles pour VTF simplifié (SAP étant une syntaxe abstraite représentant le langage VTF simplifié).

Pour visualiser quelles peuvent être ces règles de typage, nous imaginons ci-après quelques morceaux de programme quelconques et y appliquons les règles.

- **L'expression** `x+2`

  Les règles à appliquer sont celles concernant la somme entre deux expressions. On sait que le type de 2 est **int**. Il reste donc à aller voir le type de `x` là où `x` est déclaré. Ensuite, il s'agit de vérifier s'il est compatible avec le type **int** et enfin, de typer l'expression `x+2`.

  - Si `x` est déclaré de type **int**, le type de `x+2` est **int**.
  - Si `x` est déclaré de type **float**, le type de `x+2` est **float**
  - Si `x` est déclaré de type **String**, le type de `x+2` est **String**.
  - Dans tous les autres cas de figure, il y a incompatibilité de types et il y a donc erreur.

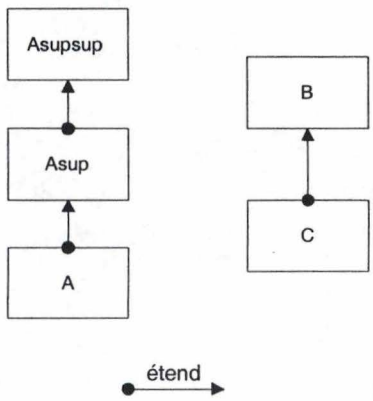
[3] Cette vérification est faite à ce niveau parce que notre parser n'en était pas capable. Celà est dû au choix de réaliser une analyse syntaxique sur une grammaire LL(1).

[4] Les règles données là reprennent à la fois les vérifications de la *forme* et des *types* d'un programme

- **L'instruction `x=y` ;**
  Les règles à appliquer sont celles concernant l'affectation. Il s'agit ici de vérifier si le type de `y` *spécialise* le type de `x`.
  En VTF simplifié, la relation de spécialisation entre les types est :
  - tout type se spécialise lui-même (réflexivité)
  - **int** spécialise **float**
  - une classe `B` spécialise une classe `A` si
    `B extends C ... extends A`

- **L'expression `(A)x`**
  Cette expression est bien typée si le type de `x` spécialise `A`. Le type de l'expression est dans ce cas `A`.

- **L'appel de méthode**
  Soit l'expression `a.foo(x,y)` avec `x` de type **int** et `y` de type `C`
  soit la hiérarchie de classes concrètes



Il faut, pour que cet appel soit valable, qu'il existe une méthode `foo` possédant deux paramètres tels que, le type du premier paramètre est spécialisé par **int** (i.e est soit **int**, soit **float**) et, le type du deuxième paramètre est spécialisé par `C` (i.e est soit `C`, soit `B`). Cette méthode doit être cherchée dans la classe `A` puis dans la classe `Asup` puis enfin dans la classe `Asupsup`. Si aucune méthode satisfaisante n'est trouvée, il y a erreur.

Il peut y avoir conflit entre plusieurs classes. Dans ce cas, on prend la méthode dont les paramètre ont le type le plus *spécialisé.*

Par exemple, si dans la classe `A` se trouvent les méthodes

```
foo(int x, B y){...}
```

et

```
foo(int x, C y){...},
```

scope	informations présentes(si le scope est complet)
`scope général`	une liste de toutes les classes du programme avec pour chaque classe, les informations suivantes : son nom, la classe parent, la liste des champs de la classe,la liste des méthodes(signature+type de retour) de la classe et la liste des constructeurs de la classe
`scope d'une classe`	la liste des champs de la classe, la liste des méthodes (signature+type de retour) de la classe et la liste des constructeurs de la classe
`scope des paramètres formels`	la liste des paramètres formels de la méthode ou du constructeur courant(e) auxquels sont associés leur type
`scope des variables locales`	la liste des variables locales du bloc en question auxquelles sont associés leur type

TABLEAU 2 : Liste des différents scope

la méthode à appliquer est la deuxième citée vu que `C` étend `B`.

Il peut cependant y avoir ambiguïté si, par exemple, les deux méthodes déclarées dans `A` sont

```
foo(int x, B y){...}
```

et

```
foo(float x, C y){...}.
```

En effet, **int** spécialise **float** mais `C` spécialise `B`. Il y a dans ce cas erreur.

4.1.3 La conception en elle-même

A partir des portées, nous définissons les *scope* (le mot signifie "portée" en anglais) comme des tables où sont associées aux différents identificateurs des informations les concernant. Ces informations portent essentiellement sur les types et sont obtenues grâce aux déclarations. Chaque déclaration ajoute une entrée dans le scope correspondant à sa portée. Le tableau 2 reprend les différents scope et leur contenu.

Tous ces scope forment une pile et chaque scope possède un pointeur vers le scope qui l'englobe (i.e vers le scope qui le précède dans la pile) (cf. Figure 4.1). Ainsi, par exemple, lorsque l'on doit rechercher certaines informations sur un identificateur afin d'en connaître le type, on peut parcourir la pile en partant de son sommet. Cette pile

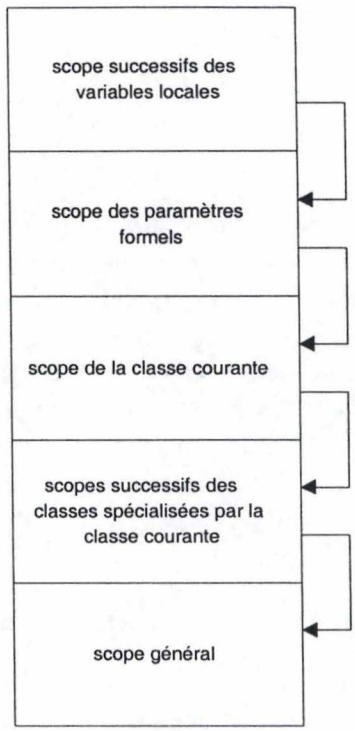


FIG. 4.1 – La pile de scope

est bien sûr dynamique c'est-à-dire qu'elle change tout le temps. Par exemple, lorsqu'on entre dans un bloc de méthode, on ajoute le scope des variables locales lui correspondant et, lorqu'on quitte cette méthode, on retire le scope de variables locales et le scope des paramètres formels corespondant à la méthode.

Donc, si l'opération consiste à "type checker" une expression qui se trouve dans le bloc d'un `if` se trouvant dans la methode `foo(...)` de la classe `A`, les scope alors présents seront :

- le scope général,
- les éventuels scope des classes spécialisées par la classe `A`,
- le scope de la classe `A`,
- le scope des paramètres formels de la méthode `foo(...)`
- le scope des variables locales du bloc de la méthode `foo(...)`,
- le scope des variables locales du bloc du `if`.

Nous avons décidé de concevoir un type checking en trois passes (i.e trois parcours de l'arbre[5]). Cette décision est due au fait que VTF simplifié est un langage non ordonné c'est-à-dire qu'un objet ne référence pas nécessairement à un objet déclaré avant lui dans

[5]les deux premiers parcours ne se font pas entièrement car ne nécessitent pas d'aller jusqu'aux feuilles

le programme. Nous expliquons ci-après le rôle assumé par chacune de ces passes.

Rôle de la première passe

Lors du premier parcours de l'arbre syntaxique, nous nous contentons de stocker toutes les classes dans le **scope général**. Aucune autre information que le nom de chacune des classes n'est pour l'instant stockée dans le scope général.

A ce niveau-ci, une vérification peut néanmoins déjà avoir lieu : pour chaque nom de classe, nous vérifions s'il n'est pas déjà présent dans le scope. S'il ne l'est pas, nous passons au deuxième parcours. S'il l'est, nous interrompons tout et annnonçons l'erreur à l'utilisateur.

Rôle de la seconde passe

A l'entame du deuxième parcours de l'arbre syntaxique, nous avons à notre disposition l'ensemble des classes présentes dans le programme. Lors de ce deuxième parcours, nous remplissons le **scope général** c'est-à-dire que nous associons à chaque classe les informations suivantes : la classe parent, la liste des champs, des méthodes et des constructeurs de la classe.

Ce deuxième parcours nous permet de vérifier si chaque classe qui est étendue par une autre l'est bien par une classe présente dans le programme.

Nous nous assurons également dans ce parcours qu'il n'est pas déclaré dans une même classe deux champs de même nom, ni deux méthodes ou deux constructeurs de même signature.

Enfin, c'est à ce niveau que nous contrôlons que les paramètres formels d'une même méthode ont bien tous des noms différents.

Rôle de la troisième passe

Le troisième parcours de l'arbre syntaxique nous permet de terminer les vérifications de cohérence et de réaliser le type checking.

Tout d'abord, nous vérifions l'absence de cycle dans l'héritage entre les différentes classes.

Nous constatons ensuite que les méthodes abstraites de chaque classe abstraite sont bien toutes étendues par la ou les éventuelle(s) classes étendant la classe abstraite en

question.

Nous pouvons alors ajouter, compléter ou supprimer les différents scope au fur et à mesure qu'on en a (plus) besoin.

Grâce à ces scope et aux informations qu'ils contiennent, nous pouvons nous assurer de l'absence de conflit entre les champs, paramètres formels et variables locales et enfin, procéder au "type checking". Ainsi nous associons à chaque identifiant et à chaque expression le type correspondant. En cas d'incompatibilité de types, le type checking n'est pas interrompu mais un type "erreur" est associé à l'expression mal typée et est ensuite propagé.

4.2 Implémentation

Les objets nécessaires au type checking sont définis dans le package `parser`. En effet, le type checking se réalise sur l'arbre abstrait obtenu par le parser. Il s'agit donc de parcourir cette arbre et, à chaque noeud syntaxique, d'exécuter certaines vérifications. Dans chaque classe représentant un noeud syntaxique est/sont donc déclarée(s) une methode `semantic(Env env,int pass )` ou/et une méthode `typeChecking(Env env,int pass)`. Les méthodes `typeChecking(Env env,int pass)` sont spécifiques aux expressions et aux identificateurs car elles renvoient un type (celui de l'expression ou identificateur en question). Le paramètre de type `Env` de ces méthodes contient les informations nécessaires à l'exécution de la méthode. Ces informations sont stockées dans une pile de *scope* (cf. section 4.1.3). Ces scope sont eux-mêmes représentés par des instances de la classe `Scope`. Cette classe est composée d'une `Hashtable` associant à des identificateurs leur type. Il s'agit d'une notion assez élargie de type puisque nous "typons" les noms de classe, les noms de méthode et les noms de constructeur en plus des variables. Ces types sont représentés par des classes et sont structurés comme le montre la figure 4.2.

Les méthodes `semantic(Env env, int pass)` et `typechecking(Env env,int pass)` font également appel à certaines méthodes se trouvant dans `Env`. Les méthodes appelées dépendent de la valeur du paramètre `pass` qui désigne quel parcours est en cours (le premier, le deuxième ou le troisième). Nous reprenons ci-dessous quelques exemples typiques de méthodes se trouvant dans `Env` et utilisées pour les vérifications.

- `void addScope()` Cette méthode est appelée quand on entre dans une classe, dans une méthode, dans un bloc... pour créer le scope correspondant et l'ajouter à la fin de la pile.

- `void rmScope()` Cette méthode supprime le scope se trouvant au sommet de la pile.

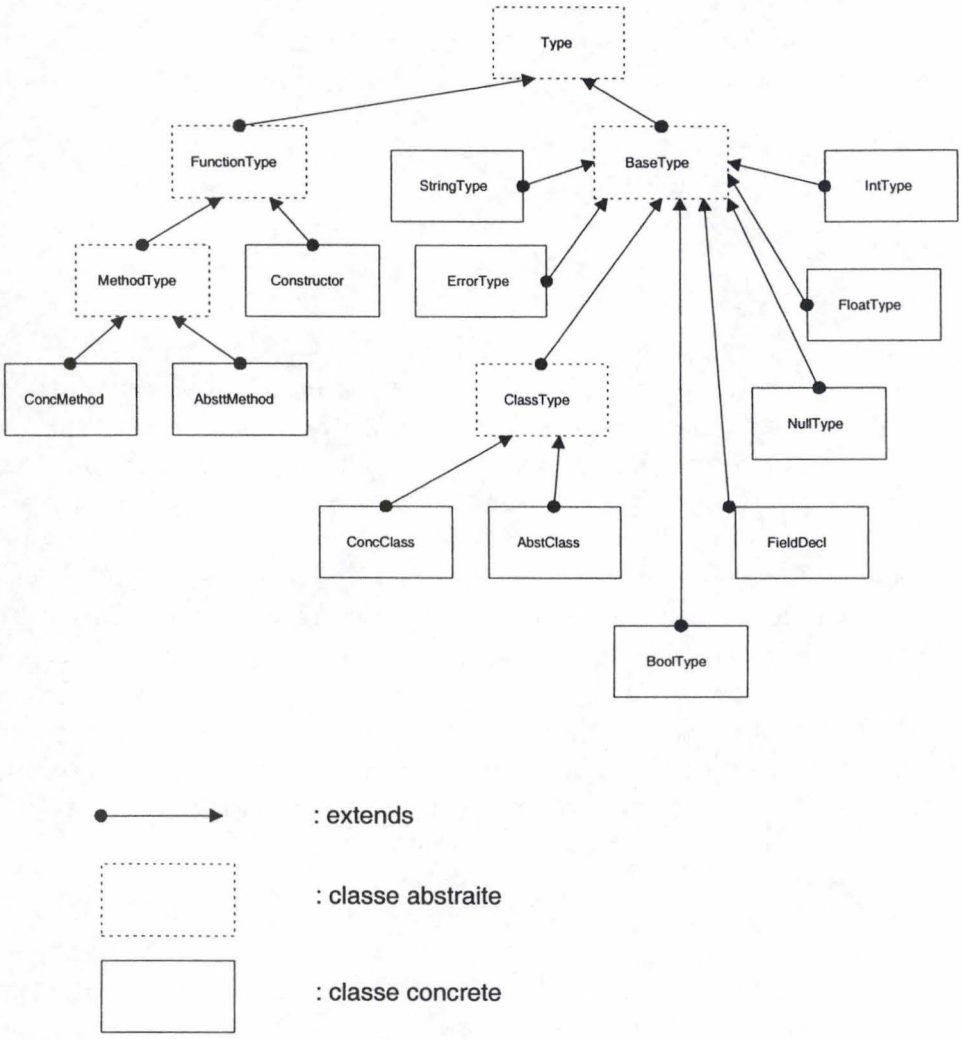


FIG. 4.2 – Structure des types

Est appelée lorsqu'on quitte un block, une classe,...

- `void addId(String id,Type type)` Cette méthode rajoute un élément dans le dernier scope de la pile. Elle est appelée lorsqu'on arrive dans une déclaration.

- `Type getType(String id)` Cette méthode recherche l'identificateur et renvoie le type qui lui est associé. La recherche commence par le sommet de la pile de scope et descend jusqu'à ce que l'identificateur soit trouvé. Si celui-ci n'est jamais trouvé, un message d'erreur est affiché et le type associé à l'expression est de type `ErrorType`

- `boolean verifCycling(String className)` Cette méthode vérifie qu'il n'y a pas de cycle entre les classes.

- `BaseType getReturnType(ClassType classType,String methodName,Vector list)` Cette méthode, très complexe, renvoie le type de retour de la méthode de nom `methodName`,dont les types des paramètres sont dans `list` et dont la classe `classType` est dans la portée. Elle est complexe car détecte s'il y a ambiguïté.

L'exécution du type checking se divise donc en trois appels successifs de la méthode `semantic(Env, int pass)` se trouvant dans la racine de l'arbre abstrait c'est à dire dans la classe `ASTProgram`. Cette méthode fait appel aux méthodes de même signature se trouvant dans les noeuds successifs de l'arbre pour finalement lors de la troisième passe, arriver aux feuilles de l'arbre et obtenir l'arbre abstrait typé.

Chapitre 5

Traduction

L'objectif de notre traducteur est de traduire l'arbre abstrait typé représentant le programme en VTF simplifié en un arbre abstrait représentant ce même programme en SAP. La différence entre les deux syntaxes abstraites est que SAP est un langage plus explicite, plus proche du langage des organigrammes. Une caractéristique fondamentale de SAP est donc qu'il fonctionne par branchement c'est-à-dire via des points de programme. Une autre différence essentielle, évoquée dans la section 1.2 est que SAP est un langage typé. Ce problème est quasiment résolu étant donné que nous travaillons à partir de l'arbre abstrait typé de VTF simplifié.

5.1 Conception

Pour expliquer les point essentiels de notre traduction nous reprenons ci-après une à une les différences par rapport à VTF simplifié mentionnées dans la section 1.2. Pour chacune de ces différences nous fournissons une brève explication de la façon dont nous procédons à la transformation.

- Présence de labels en début d'instruction et éventuellement à la fin pour identifier l'instruction suivante

  Le label doit être différent pour toutes les instructions du programme. Nous avons décidé que ce label serait composé du nom de la classe, de la signature de la méthode et d'un numéro identifiant l'instruction. Cette décision s'est faite pour de simple raison de facilité au niveau de l'implémentation.

- Présence d'un label associé à chaque programme, désignant la première instruction de la méthode principale

Pour identifier la méthode principale en VTF simplifié, nous demandons, par convention, au programmeur d'appeler `main` la méthode par laquelle il veut commencer son programme. Le label du programme en SAP correspond donc au label de la première instruction de cette méthode `main`.

- Présence obligatoire d'au moins un constructeur explicite par classe

  Si un constructeur est déjà présent dans la classe en VTF, rien n'est changé, sinon un constructeur vide est ajouté.

- Impossibilité d'initialiser un champ ou une variable locale

  Cette contrainte nous oblige, si un champ est initialisé dans le programme VTF simplifié, d'ajouter en SAP une instruction au début de chaque constructeur (même le constructeur par défaut).
  Par exemple,

```
class A{
    int x=3;
    A(int j){j=3;}
}
```

devient

```
class A{
    int x;
    A(){
        0   x=3 1;
        1   return;
    }
    A(int j){
        0   x=3 1;
        1   j=3 2;
        2   return;
    }
}
```

  Pour les déclarations de variables, une instruction est simplement ajoutée en début de programme.

- Disparition de la notion de blocs imbriqués et par conséquent, obligation de nommer différemment les variables locales d'une même méthode

  Le renommage se fait au niveau de l'arbre VTF simplifié où un second nom est

associé aux identifcateurs désignant des variables locales.

- Obligation de terminer chaque méthode par une instruction `return`

  A la fin de chaque méthode **void** et de chaque constructeur est ajoutée une instruction **return**.

- Absence des boucles et des tests qui doivent être traduits en utilisant les branchements

  Par exemple,

```
x=0;
while(x<=2){
    x=x+1;
}
return x;
```

  devient

```
0   x=0          1;
1   if x<=2 2    3;
2   x=x+1        1;
3   return x;
```

- Obligation de passer par une variable intermédiaire lors des appels de fonction utilisées à l'intérieur d'une expression quelconque

  Des variables intermédiaires doivent être créées. Par exemple, reprenons un morceau du programme de la section 1.3 :

```
b=exprLeft.operation(5)!=exprRight.operation(7)
```

  devient

```
int v4;
int v5;
...
4   v4=exprLeft.operation(5)     5;
5   v5=exprLeft.operation(7)     6;
6   b=v4!=v5;
...
```

5.2 Implémentation

Les différents éléments de l'arbre abstrait auxquels aboutit le traducteur se trouvent dans le package `abstractSyntax`. Chacune des classes sensées représenter les noeuds de

l'arbre résultat contient une méthode `traduction` prenant comme argument la partie de l'arbre VTF correspondant au noeud en question. Le traducteur est lancé en créant une instance de la classe `ASTProgramSAP` qui prend en argument l'instance de `ASTProgram` représentant le programme VTF simplifié. La méthode `traduction(ASTProgram)` de cette classe est alors appelée. Cette méthode appelle les méthodes `traduction(ASTDefClass)` des classes composant le programme en prenant chaque fois comme argument la classe correspondante. La traduction est lancée et les changements sont réalisés chaque fois que l'on arrive à un noeud concerné.

Deuxième partie

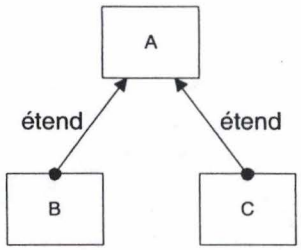
Interprétation abstraite d'un sous-langage de SAP

Introduction

Le type checking tel que nous l'avons réalisé sur le langage VTF simplifié puis reporté à SAP consiste en une vérification des types statiques. Cette vérification lie aux variables leur type déduit de leur déclaration. Cependant, dans un langage Orienté Objet, le type dynamique (i.e lors de l'exécution) d'une variable peut être différent du type déclaré de cette variable puisqu'il peut être une spécialisation de ce dernier. Un programme bien typé au niveau statique peut donc s'avérer "mal typé" lorsqu'il est exécuté.

Exemple :

Etant donné la hiérarchie de classes,



considérons la suite d'instructions suivante :

```
...
A x=new B();
...
C y=(C)x;
...
```

Manifestement, une analyse des types statiques accepte ce code car elle considère `x` comme étant de type `A`. Par contre, une analyse optimale des types dynamiques associe à `x` le type `B` et nous permet de déceler l'invalidité du cast.

Il peut donc être intéressant de disposer d'une information plus précise au sujet des types réels des variables que celle fournie par le type checking. C'est pourquoi dans cette seconde partie, nous réalisons une analyse des types dynamiques.

Pour réaliser ces vérifications de types dynamiques sans exécuter le programme, nous nous basons sur une méthodologie particulière appelée *interprétation abstraite.*

Cette partie commence donc tout naturellement par une introduction à l'interprétation abstraite appliquée à la vérification de types. Ensuite, nous définissons un nouveau langage quelque peu restreint par rapport à SAP car appliquer la théorie de l'interprétation abstraite à SAP dépasse le cadre de ce mémoire. Nous définissons donc successivement la syntaxe abstraite de ce langage (en soulignant les différences importantes par rapport à SAP) et la sémantique dénotationnelle de ce même langage. Nous appliquons ensuite à l'analyse des types de notre langage les concepts intervenant dans la théorie de l'interprétation abstraite. Enfin, nous donnons quelques résultats de notre analyse des types utilisant l'interprétation abstraite.

Chapitre 6

Introduction à l'interprétation abstraite

Comme nous l'avons dit, l'objectif de ce travail est de procéder à une analyse des types dynamiques sans toutefois exécuter le programme. Il s'agit donc de réaliser l'analyse statique des types dynamiques du programme.

Or, on sait que calculer des propriétés sémantiques de progammes est un problème indécidable (théorème de Rice). L'idée est de travailler avec des approximations que nous appelons des abstractions.

Le principe de l'interprétation abstraite est de réaliser une *exécution abstraite* d'un programme. Il s'agit de simuler l'ensemble des exécutions concrètes du programme sur un domaine *abstrait* plutôt que de l'exécuter sur le domaine de calcul normal. Les éléments de ce domaine abstrait primitif représentent certaines propriétés des éléments du domaine normal. On peut définir le lien entre ces domaines par une fonction d'abstraction et un fonction de concrétisation comme suit :

Soit C le domaine de calcul normal, A est un domaine abstrait primitif possible si et seulement si il existe une fonction d'abstraction $Abs : C \rightarrow A$ telle que $Abs(c)$ est une propriété de c ou, en d'autres termes il existe une fonction de concrétisation $Cc : A \rightarrow \mathcal{P}(C)$ telle que $Cc(a) = \{c : c \text{ vérifie } a\}$

Pour appliquer cette théorie, il est nécessaire de définir une sémantique abstraite du langage. Avant de définir cette sémantique abstraite, nous devons définir une sémantique concrète du langage. Pour ce faire, il s'agit de définir des domaines sémantiques représentant les états possibles d'un programme et des fonctions de transition entre ces états. Ces transitions correspondent aux instructions. De cette façon, l'exécution d'un programme peut être assimilée à une suite de transitions entre des états concrets σ_i.

Le principe est d'ensuite décider d'une abstraction de ces états c'est-à-dire d'un domaine abstrait. De nombreuses représentations abstraites des états sont possibles et ne

dépendent pas nécessairement du domaine primitif [1]. La représentation des états abstraits est une simplification de la représentation des états concrets. Dans notre cas, nous nous imposons que le domaine abstrait soit fini. Une fois ce domaine défini, nous pouvons réaliser l'exécution abstraite du programme qui consiste en une suite de transitions entre des états abstraits.

Moyennant certaines conditions, notamment sur le domaine primitif, on peut ensuite calculer par un algorithme du point fixe l'ensemble des états abstraits α tels que

$$\alpha_{init} \rightarrow^* \alpha$$

où α_{init} est l'état abstrait initial du programme.

Cela signifie que, si l'on travaille sur un langage des organigrammes, des informations peuvent être apportées à chaque point de programme par lequel l'exécution abstraite est passée. Ces informations portent sur l'état du programme et sont étroitement liées au domaine abstrait choisi. Il peut, par exemple, s'agir d'informations sur les types.

[1] Le domaine abstrait primitif est d'ailleurs souvent défini après la représentation abstraite des états. Cette dernière se fait alors sur base d'un domaine abstrait générique

Chapitre 7

Définition d'un sous-langage de SAP

7.1 Modification par rapport à SAP

Les principales modifications du nouveau langage par rapport à SAP sont les suivantes.

1. Les notions de classe abstraite et de méthode abstraite sont absentes.

2. Il n'y a plus de constructeurs explicites. Toute instance dans ce nouveau langage est créée à partir du constructeur par défaut de la classe qu'elle instancie.

3. Les opérateurs `||`, `&&` , `!` et l'opposé ne sont plus considérés.

4. Les types de base **float** et **boolean** ne sont plus considérés.

5. Certaines fonctions ont un nom qui leur procure un statut particulier : **toString**(*nvar*), **read**(*nvar*), **write**(*nvar*), **toInt**(*nvar*), *nvar*.**length**(), *nvar*.**compareTo**(*nvar*), *nvar*.**equals**(*nvar*) (avec *nvar* variable locale). Toute ces fonctions peuvent être utilisées sans avoir été déclarées auparavant (et ne peuvent d'ailleurs pas être déclarées) et une signification leur est conférée. Ce sont essentiellement des fonctions utilisées sur des String. En effet, le type String est un intermédiaire entre un type de base et un type classe. Sa sémantique est expliquée dans la section 2.2.3.2.

6. Tout appel de méthode ne peut avoir comme paramètres effectifs que des variables locales et non des expressions quelconques.

7. On ne peut pas affecter à un champ autre chose qu'une variable locale.

8. Le "type" *void* n'existe plus et donc toute méthode doit renvoyer un résultat, mis à part la méthode *main* qui ne peut rien renvoyer.

9. Aucune instruction **return**(*nvar*)ne peut renvoyer une autre expression qu'une variable locale.

10. On ne peut juste qu'accéder aux champs de l'instance courante.

11. Il y a une contrainte d'unicité supplémentaire sur les champs : si on a par exemple les fragments de code suivants :

```
class A extends B {...}
class B extends C {...}
class C extends D {...}
```

    les champs déclaré dans `A, B, C, D` doivent tous avoir un nom différent.

12. Il est interdit de surcharger des noms de méthodes. Une méthode peut être redéclarée dans une classe à condition de conserver la même signature (types des paramètres) et le même type résultat que la méthode de même nom déclarée dans la classe qu'elle spécialise.

7.2 Syntaxe abstraite du langage

Les domaines syntaxiques et la syntaxe abstraite du langage sont repris dans la figure 7.1.

7.3 Sémantique opérationnelle du langage

Pour pouvoir faire notre analyse statique des types en utilisant l'interprétation abstraite, nous devons définir une sémantique abstraite du langage. Avant de définir cette sémantique abstraite nous devons définir une sémantique concrète.Cette sémantique concrète est nécessaire pour s'assurer de la correction de la sémantique abstraite.

Domaines syntaxiques et variables génériques

$nclass \in \mathbb{N}class$	: Noms de classe
$nfield \in \mathbb{N}field$	: Noms de champ
$nmethod \in \mathbb{N}method$	: Noms de methode
$nparam \in \mathbb{N}param$	: Noms de paramètre formel
$nvar \in \mathbb{N}var$	: Noms de variable locale
$lit \in \mathbb{L}it$	: Littéraux
$p, q, r \in \mathbb{P}ts$	: Points de programme (labels)
$i \in \mathbb{Z}$	: Valeurs entières
$str \in \mathbb{C}har*$	: Strings
$type \in \mathbb{T}ype$	: Types
$declfield \in \mathbb{D}eclfield$	: Déclarations de champ
$declmethod \in \mathbb{D}eclmethod$	: Déclarations de méthode
$instr \in \mathbb{I}nstr$	: Instructions
$cond \in \mathbb{C}ond$	: Conditions
$expr \in \mathbb{E}xpr$	: Expressions
$cop \in \mathbb{C}op$	: Opérateurs de comparaison
$aop \in Aop$	: Opérateurs arithmétiques
$prog \in \mathbb{P}rog$	: Programmes

Syntaxe abstraite

prog	$::=$ $defclass^{+} lab$
defclass	$::=$ **class** *nclass* [**extends** *nclass*] $declfield^{*}$ $declmethod^{*}$
declfield	$::=$ *type nfield*
type	$::=$ **int** | **String** | *nclass*
declmethod	$::=$ **method** *type nmethod declparam* declvar* instr**
instr	$::=$ *lab nfield* = *nvar*
	| *lab nvar* = *expr*
	| *lab nvar* = *new nclass*()
	| *lab* **if** *cond lab lab*
	| *lab return nvar*
	| *lab return*
	| *lab nvar* =**toString**(*nvar*)
	| *lab* **read**((*nvar*))
	| *lab* **write**((*nvar*))
	| *lab nvar* = *cible.nmethod*(*nvar**)
cible	$::=$ **this** | **super** | *nvar*
cond	$::=$ *nvar* **instanceOf** *nclass* | *nvar cop nvar*
cop	$::= == \mid != \mid < \mid > \mid \leq \mid \geq$
expr	$::=$ *nfield* | *nparam* | *nvar* | *nvar op nvar*
	| *lit* | (*nclass*)*nvar* | **this**
	| **toInt(** *nvar* **)** | *nvar*.**length()**
	| *nvar*.**compareTo(** *nvar* **)**
	| *nvar*.**equals(** *nvar* **)**
aop	$::= + \mid - \mid * \mid / \mid \%$
lit	$::=$ **null** | *i* | *str*

FIG. 7.1 – Syntaxe abstraite du sous-langage

7.3.1 Notions préliminaires

Pour définir notre sémantique concrète, nous commençons par préciser quelques notions relatives à notre langage.

Noms effectivement utilisés

Les ensembles de noms $\mathbb{N}class$, $\mathbb{N}field$, $\mathbb{N}param$, $\mathbb{N}var$, $\mathbb{N}method$, sont infinis et disjoints. Dans un programme donné, seul un sous-ensemble fini de chacun de ces ensembles est utilisé. Cependant, par abus de langage, nous exprimons les ensembles des noms relatifs à un programme par ces mêmes notations.

Unicité des noms dans le programme

Il y a un certain nombre de contraintes d'unicité évidentes pour un programme donné. Par exemple, les classes doivent avoir des noms différents. Ces contraintes ne sont pas détaillées ici. Elles sont du même type que celles exprimées au point 1.6.1.2 pour VTF simplifié.

Relation d'extension : $extends \ \mathbb{N}class \times \mathbb{N}class$

$nclass_1$ *extends* $nclass_2$ si et seulement si la déclaration de la classe $nclass_1$ commence par **class** $nclass_1$ **extends** $nclass_2$ ...

La relation d'extension ne peut pas contenir de cycle. Autrement dit, il n'existe pas $n \geq 2$ tel que $nclass_1, ..., nclass_n \in \mathbb{N}class$ et $nclass_1$ *extends* $nclass_2$ ... *extends* $nclass_n$ et $nclass_1 = nclass_2$

Relation de spécialisation : $\preceq \ \mathbb{N}class \times \mathbb{N}class$

La relation de spécialisation est la plus petite relation d'ordre qui contient la relation d'extension.

Par exemple, si $nclass_1$ *extends* $nclass_2$ et $nclass_2$ *extends* $nclass_3$
alors $nclass_1 \preceq nclass_2$, $nclass_1 \preceq nclass_3$, $nclass_2 \preceq nclass_3$

Super-classe d'une classe : $super : \mathbb{N}class \nrightarrow \mathbb{N}class$

$super(nclass) = nclass'$ si et seulement si $nclass$ *extends* $nclass'$

Ensemble des champs d'une classe : $fields : \mathbb{N}class \rightarrow \mathcal{P}(\mathbb{N}field)$

Soit $nclass_1$, $nclass_2$, ..., $nclass_n \in \mathbb{N}class$ tels que
$nclass_1$ *extends* $nclass_2$ ... *extends* $nclass_n$ et $nclass_n$ est une classe primitive ($super(nclass_n)$ n'est pas définie)
Par définition $fields(nclass_1)$ est l'ensemble des champs déclarés dans
$nclass_1$, $nclass_2$, ..., $nclass_n$ (pour rappel, ces champs doivent tous avoir des noms différents).

Instruction débutant à un point de programme :

$$findInstr \ : \ \mathbb{P}ts \ \rightarrow \ \mathbb{I}nstr$$

Un point de programme utilisé par le programme est l'étiquette d'une et une seule instruction. Donc, à chaque point de programme $p \in \mathbb{P}ts$ correspond l'instruction $findInstr(p)$ en question.

La bonne méthode d'une classe donnée :

$$findMeth \ : \ \mathbb{N}class \ \times \ \mathbb{N}method \ \nrightarrow \ \mathbb{D}eclmethod$$

Au préalable, rappelons que la surcharge de méthode est interdite dans ce langage.

$findMeth(nclass, nmethod) = declmethod$ où
- $declmethod$ figure dans $defclass$ = **class** $nclass_i$ ... et
- $nclass$ = $nclass_1$ *extends* $nclass_2$ ... *extends* $nclass_i$ ... et
- $declmethod$ est une déclaration de méthode de nom $nmethod$ et
- il n'y en a pas une de même nom pour $nclass_1$, ..., $nclass_{i-1}$

7.3.2 Domaines sémantiques

Avant de réaliser la sémantique opérationnelle du langage, nous devons définir d'autres ensembles que les domaines syntaxiques.

- $\mathbb{L}oc$ **: ensemble des "locations" (emplacements en mémoire)**
  Une location représente l'emplacement en mémoire d'un objet.

- $\mathbb{V}al = \mathbb{Z} + \mathbb{L}oc + \{\textbf{null}\}$ **: ensemble des valeurs**
  La valeur d'une variable est soit une valeur entière, soit l'adresse d'une instance, soit **null** lorsqu'aucune adresse ne lui est associée.

- $\mathbb{I}nst = \mathbb{N}class \times (\mathbb{N}field \nrightarrow \mathbb{V}al) + \{\textbf{String}\} \times \mathbb{C}har^*$ **: ensemble des instances**

  Une instance est
  - soit une instance de classe composée de son type (nom de sa classe) et d'une fonction allant de l'ensemble des champs de sa classe dans l'ensemble des valeurs,
  - soit une instance de string.

Les string ont, dans ce langage, un statut particulier. D'une part, contrairement aux entiers, leur type n'est pas un type de base. D'autre part, il ne s'agit pas non plus d'instances de classe puisqu'il n'existe pas, dans ce langage de classes prédéfinies. La valeur d'une variable de type string est donc un pointeur **null** ou un pointeur vers le couple <**String**,s> où s est la chaîne de caractères qui forme le string.

- $\mathbb{E}nv = (\mathbb{N}param + \mathbb{N}var) \rightarrowtail \mathbb{V}al + \{\textbf{noninit}\})$
  Un environnement représente l'état (valeurs) des paramètres et des variables locales de la méthode courante.

- $\mathbb{P}ile = (\mathbb{P}ts \times \mathbb{L}oc \times \mathbb{E}nv)^*$
  La pile des appels "suspendus" de méthode "mémorise" pour chaque appel, le point de programme précédant l'appel, la référence de l'objet "appelant", et l'environnement de la méthode appelante.
  L'idée est d'avoir une structure qui nous permet, lorsqu'on se trouve au niveau d'un appel de méthode, de sauver l'état courant avant d'exécuter la méthode en question.
  En fait, cette pile se remplit lorsqu'on est face à l'instruction $nvar_2 = nvar_1.nmethod()$ et se vide d'un élément (son sommet) lorsqu'on est face à l'instruction *return nvar* (i.e lorsqu'on retourne à la méthode appelante).

- $\mathbb{S}tore = \mathbb{L}oc \rightarrowtail \mathbb{I}nst$
  La mémoire (ou store) associe à chaque référence valide (ayant fait l'objet d'une allocation de mémoire) l'instance qui lui correspond (qu'elle identifie).

- $\mathbb{E}tat = \mathbb{P}ts \times \mathbb{L}oc \times \mathbb{E}nv \times \mathbb{S}tore \times \mathbb{P}ile \times \mathbb{I} \times \mathbb{O}$ avec :
  $\mathbb{I} = (\mathbb{C}har^*)^*$
  $\mathbb{O} = (\mathbb{C}har^*)^*$

  Un état possible de l'exécution d'un programme se compose :
  - du point de programme courant : il identifie la prochaine instruction à exécuter ;
  - de la location courante : elle identifie l'instance courante ;
  - de l'environnment courant (de la méthode en cours d'exécution) ;
  - de l'état courant de la mémoire ;
  - de la pile des appels suspendus ;
  - de l'état de l'entrée standard ;
  - de l'état de la sortie standard.

7.3.3 Fonctions sémantiques

Choix d'une nouvelle location : $newLoc : \mathbb{S}tore \rightarrow \mathbb{L}oc$

Cette fonction vérifie l'"axiome" : $\forall\ s\ \in\ \mathbb{S}tore\ :\ newLoc(s)\ \notin\ dom(s)$

Instance originelle d'une classe : $newIns\ :\ \mathbb{N}class\ \rightarrow\ (\mathbb{N}field\ \rightarrowtail\ \mathbb{V}al)$

Cette fonction est déterminée par le programme.

Soient $nclass\ \in\ \mathbb{N}class$ et $nfield\ \in\ fields(nclass)$,
$(newIns\ nclass\ nfield)$ = **null** si $nfield$ est de type classe ou **String**
0 si $nfield$ est de type **int**

Etat initial de la mémoire : $s_0\ \in\ \mathbb{S}tore$

Soit $s_\emptyset$, la mémoire vide telle que $s_\emptyset\ \in\ \mathbb{S}tore$ et $dom(s_\emptyset)\ =\ \{\}$,
soit <**Main**,$f_\emptyset$>, l'instance vide telle que $f_\emptyset\ \in\ \mathbb{N}field\ \rightarrowtail\ \mathbb{V}al$ et $dom(f_\emptyset = \{\}$.

Soient $lit_1, ..., Lit_n$ $(n \geq 0)$, les littéraux de type **String** du programme.
Ils définissent n strings $str_1, ..., Str_n$.
Posons :

$$l_0\ =\ newLoc(s_\emptyset)\ ;\ \ s_1\ =\ s_\emptyset[l_0/ < \textbf{Main}, f_\emptyset >]$$

$$l_i <=\ newLoc(s_i)\ ;\ \ s_{i+1}\ =\ s_i[l_i/ < \textbf{String}, str_i >]\ (1 \leq i \leq n)$$

Par définition, $s_0 = s_{n+1}$

Mise à jour d'un champ : $update : \mathbb{N}field \rightarrow \mathbb{L}oc \rightarrow \mathbb{V}al \rightarrow \mathbb{S}tore \rightarrow \mathbb{S}tore$

Lorsqu'on met à jour un champ, on modifie le store. On a comme données l'instance courante $s(l)$, le nom du champ que l'on veut modifier $(nfield)$ et la valeur que l'on veut lui donner (v). Le nouveau store est donc le même que le précédant sauf dans l'instance courante où la valeur de $nfield$ a changé.

$update\ nfield\ l\ v\ s\ =\ s'$ où $dom(s')\ =\ dom(s)$ et
$s'(l')\ =\ s(l')\ \forall l'\ \in\ dom(s)\backslash\{l\}$ et
$s'(l)\ =\ s(l)[nfield/v]$

Note : $< nclass, f > [nfield/v]\ =_{def} < nclass, f[nfield/v] >$

Sémantique des opérateurs

Cette sémantique détermine la valeur d'une opération arithmétique lorsqu'on connaît la valeur des opérandes.

L'évaluation d'une opération arithmétique donne comme résultat un entier si les valeurs des opérandes sont des entiers. Dans tous les autres cas, l'évaluation de l'opération est une erreur sauf pour l'opération d'addition.

Si l'un des deux opérandes est un string, la valeur d'une somme est en effet la concaténation entre le string et la représentation sous forme d'une chaîne de caractères de l'autre valeur (peut être une location, un entier ou **null**).

Signature

$\mathcal{O} : Aop \rightarrow (\mathbb{V}al + \{\textbf{noninit}\}) \rightarrow (\mathbb{V}al + \{\textbf{noninit}\}) \rightarrow \mathbb{S}tore \rightarrow \mathbb{V}al + \{\textbf{erreur}\}$

L'addition

$$\mathcal{O}[\![+]\!]\ v_1\ v_2\ s \quad = \quad \begin{array}{ll} v_1 + v_2 & \underline{\text{si}}\ v_1, v_2 \in \mathbb{Z} \\ \textbf{erreur} & \underline{\text{sinon}} \end{array}$$

Les autres opérations arithmétiques

même principe que l'addition.

Les opérations d'égalité et d'inégalité donnent une valeur pour n'importe quelles opérandes initialisées. Si ces opérandes sont des String ou des classes, il s'agit d'une égalité de pointeurs. Les opérateurs $\leq$, $\geq$, $<$, $>$ ne peuvent être utilisés qu'entre des entiers.

Signature

$\mathcal{C} : \mathbb{C}op \rightarrow (\mathbb{V}al + \{\textbf{noninit}\}) \rightarrow (\mathbb{V}al + \{\textbf{noninit}\}) \rightarrow \mathbb{S}tore \rightarrow \mathbb{V}al$

L'égalité

$$\mathcal{C}[\![==]\!]\ v_1\ v_2\ s \quad = \quad \begin{array}{ll} \textbf{true} & \underline{\text{si}}\ v_1 = v_2 \neq \textbf{noninit} \\ \textbf{false} & \underline{\text{si}}\ v_1 \neq \textbf{noninit}, v_2 \neq \textbf{noninit}, v_1 \neq v_2 \\ \textbf{erreur} & \underline{\text{sinon}} \end{array}$$

L'inégalité

$$\mathcal{C}[\![!=]\!]\ v_1\ v_2\ s \quad = \quad \begin{array}{ll} \textbf{false} & \underline{\text{si}}\ v_1 = v_2 \neq \textbf{noninit} \\ \textbf{true} & \underline{\text{si}}\ v_1 \neq \textbf{noninit}, v_2 \neq \textbf{noninit}, v_1 \neq v_2 \\ \textbf{erreur} & \underline{\text{sinon}} \end{array}$$

Les comparaisons (strictes)

$$\mathcal{C}[\![\leq]\!]\ v_1\ v_2\ s \quad = \quad \begin{array}{ll} \textbf{true} & \underline{\text{si}}\ v_1, v_2 \in \mathbb{Z}\ \underline{\text{et}}\ v_1 \leq v_2 \\ \textbf{false} & \underline{\text{si}}\ v_1, v_2 \in zit\ \underline{\text{et}}\ v_1 > v_2 \\ \textbf{erreur} & \underline{\text{sinon}} \end{array}$$

$$\mathcal{C}[\![\geq]\!]\ v_1\ v_2\ s \quad = \quad \begin{array}{ll} \textbf{true} & \underline{\text{si}}\ v_1, v_2 \in zit\ \underline{\text{et}}\ v_1 \geq v_2 \\ \textbf{false} & \underline{\text{si}}\ v_1, v_2 \in zit\ \underline{\text{et}}\ v_1 < v_2 \\ \textbf{erreur} & \underline{\text{sinon}} \end{array}$$

Evaluation d'un littéral

Signature
$\mathcal{L} : \mathbb{L}it \rightarrow \mathbb{S}tore \rightarrow \mathbb{V}al$
$\mathcal{L}[\![\,\textbf{null}\,]\!]\,s = \textbf{null}$

Les entiers
$\mathcal{L}[\![\,i\,]\!]\,s = i$

Les string
La valeur d'un string est la location dont la projection 1 de l'image dans le store est le mot clé **String** et dont la projection 2 est la chaîne de caractères correspondant à ce string.
$\mathcal{L}[\![\,str\,]\!]\,s = l$ où $s(l) = <\textbf{String}, str>$

Evaluation d'une expression

Nous expliquons particulièrement les évaluations d'un champ, d'un paramètre, d'un *cast* et de l'opération **toInt**(*nvar*). Les autres sont triviales ou sont similaires à une des expressions expliquées.

Signature
$\mathcal{V} : \mathbb{E}xpr \rightarrow \mathbb{E}nv \rightarrow \mathbb{S}tore \rightarrow \mathbb{L}oc \rightarrow \mathbb{V}al + \{\textbf{erreur}\}$

Les champs
Une des restrictions de ce langage est qu'on ne peut considérer que les champs de l'instance courante. Pour évaluer un champ, on en cherche donc toujours la valeur dans l'instance courante.
$\mathcal{V}[\![\,nfield\,]\!]\,e\,s\,l = f[\![\,nfield\,]\!]$ où $< nclass, f >= s(l)$

Les paramètres
La valeur d'un paramètre se trouve dans l'environnement. Si le paramètre n'est pas initialisé, il y a erreur.
$\mathcal{V}[\![\,nparam\,]\!]\,e\,s\,l = e[\![\,nparam\,]\!]$ si $e[\![\,nparam\,]\!] \neq$ **noninit**
erreur sinon

Les variables locales
$\mathcal{V}[\![\,nvar\,]\!]\,e\,s\,l = e[\![\,nvar\,]\!]$ si $e[\![\,nvar\,]\!] \neq$ **noninit**
erreur sinon

Les opérations

$\mathcal{V}[\![nvar_1\ aop\ nvar_2]\!]\ e\ s\ l = \mathcal{O}[\![aop]\!]\ (e[\![nvar_1]\!])(e[\![nvar_2]\!])$

Les littéraux

$\mathcal{V}[\![lit]\!]\ e\ s\ l = \mathcal{L}[\![lit]\!]$

Les variables castées

Un cast (*nclass*) n'est possible sur une variable *nvar* que si le type de celle-ci spécialise le type *nclass*. Pour cela, il faut que la valeur de *nvar* se trouvant dans l'environnement *e* soit une location dont la projection 1 de l'image dans le store soit l'identificateur *nclass'* tel que $nclass' \preceq nclass$. Sinon le cast est impossible et il y a erreur.

$\mathcal{V}[\![(nclass)nvar]\!]\ e\ s\ l = e[\![nvar]\!]$ si $e[\![nvar]\!] \in \mathbb{L}oc$ et
$s(l) =< nclass', f >$ et
$nclass' \preceq nclass$
erreur sinon

Le désignateur d'instance this

$\mathcal{V}[\![\textbf{this}]\!]\ e\ s\ l = l$

Transformer un string en entier

Cette opération ne peut se faire que sur les string. Le résultat est alors la représentation en entier du string.

$\mathcal{V}[\![\textbf{toInt}(nvar)]\!]\ e\ s\ l = i$ si $s(e[\![X]\!]) =<\textbf{String}, str >$ et
str représente l'entier $i \in \mathbb{E}$
erreur sinon

Trouver la longueur d'un string

$\mathcal{V}[\![nvar.\textbf{length()}]\!]\ e\ s\ l =$ i si $s(e[\![nvar]\!]) =<\textbf{String}, str >$ et
str est de longueur *i*
erreur sinon

Comparer la longueur de deux string

$\mathcal{V}[\![nvar_1.\textbf{compareTo}(nvar_2)]\!]\ e\ s\ l =$
1 si $str_1 > str_2$
0 si $str_1 = str_2$
-1 si $str_1 < str_2$
erreur si str_1 ou str_2 n'est pas défini
où $s(e[\![nvar_i]\!]) =<\textbf{String}, str_i > (i = 1, 2)$

Evaluation d'une condition

Signature

$\mathcal{B} : \mathbb{C}ond \rightarrow \mathbb{E}nv \rightarrow \mathbb{S}tore \rightarrow \mathbb{L}oc \rightarrow \mathbb{B}ool + \{\textbf{erreur}\}$
avec $\mathbb{B}ool = \{\textbf{false},\textbf{true}\}$

Quel est le type de l'instance ?
L'opération **instanceOf** entre une variable *nvar* et un nom de classe *nclass* renvoie **true** si *nvar* est une instance de type spécialisant *nclass* et renvoie **false** sinon. Si *nvar* n'est pas de type classe (est de type **int**, **null** ou **string**), il y a erreur.

$\mathcal{B}[\![nvar \textbf{ instanceOf } nclass]\!]\ e\ s\ l =$ **true** si $nclass' \preceq nclass$
erreur si $nclass'$ n'est pas définie
où $e[\![nvar]\!] = l' \in \mathbb{L}oc$ et
$s(l') =< nclass', f >$
false sinon

Les comparaisons
$\mathcal{B}[\![X_1\ cop\ nvar_2]\!]\ e\ s\ l = \mathcal{C}[\![cop]\!]\ (e[\![nvar_1]\!])(e[\![nvar_2]\!])$

L'égalité entre deux string
Cette fonction ne peut comparer que des string. Il s'agit d'une égalité de valeurs et pas de pointeurs.

$\mathcal{B}[\![nvar_1\textbf{.equals}(nvar_2)]\!]\ e\ s\ l =$ **true** si $str_1 = str_2$
et $e[\![nvar_i]\!] = l_i \in \mathbb{L}oc\ (i = 1,2)$
$s(l_i) = <\textbf{String}, str_i>\ (i = 1,2)$
false si $str1 \neq str_2$
et $e[\![nvar_i]\!] = l_i \in \mathbb{L}oc\ (i = 1,2)$
$s(l_i) = <\textbf{String}, str_i>\ (i = 1,2)$
erreur sinon

Evaluation du type d'une variable *cible*

Nous appelons *cible* soit **this**, soit **super**, soit une variable, du moment que cette *cible* se trouve dans une instruction $nvar = cible.nmethod(nvar_1 ... nvar_m)$. Ces variables forment l'ensemble $\mathbb{C}ible$. Si c'est une variable, le type de cette variable doit bien sûr être une location pointant vers une instance de classe.

Signature
$\mathcal{T} : \mathbb{C}ible \rightarrow \mathbb{E}nv \rightarrow \mathbb{S}tore \rightarrow \mathbb{L}oc \rightarrow \mathbb{C} \times \mathbb{L}oc + \{\textbf{erreur}\}$

La cible this
$\mathcal{T}[\![\textbf{this}]\!]\ e\ s\ l =< nclass, l >$ où $s(l) =< nclass, f >$

La cible super
$\mathcal{T}[\![\textbf{super}]\!]\ e\ s\ l =< super(nclass), l >$ où $s(l) =< nclass, f >$

si $super(nclass)$ est définie
erreur sinon

Une cible variable locale

$\mathcal{T}[\![nvar]\!]\ e\ s\ l \quad =< nclass', l' > \underline{\text{si}}\ e[\![nvar]\!] = l' \in \mathbb{L}oc\ \underline{\text{et}}\ s(l') =< nclass', f' >$
erreur sinon

7.3.4 Règles de transition

L'exécution d'une instruction d'un programme peut être considérée comme une transition entre un état du programme et un autre. De cette façon, l'exécution d'un programme est assimilée à une suite de transitions d'état à état démarrant par l'état initial. Ci-après nous définissons successivement l'état initial d'un programme et, pour chaque type d'instruction, la règle à suivre par la transition correspondante.

Une règle de transition est exprimée selon le modèle :

- $\{p\}\ instr\ \{q\}$

$$< p, l, e, s, \pi, I, O > \rightarrow < q, l', e', s', \pi', I', O' >$$

où
- p représente le point de programme courant ;
- $instr = findInstr(p)$;
- q représente le point de programme suivant (absent pour l'instruction *return*) ;
- $< p, l, e, s, \pi, I, O >$ est l'état du programme avant la transition ;
- $< q, l', e', s', \pi', I', O' >$ est l'état du programme après la transition [1] ($< O >$ pour l'instruction *return*).

Ces règles de transitions utilise les fonctions définies dans la section 7.3.3. Nous donnons quelques explications pour les transition qui nous semblent les plus complexes.

Etat initial

C'est l'état $< p_0, l_0, e_0, s_0, \pi_0, I_0, O_0 >$ où

[1] Pour les composantes non modifiées, le "prime" est absent et pour les autres, les modifications sont définies.

- p_0 est le point d'entrée de la méthode **main** (i.e le label du programme) ;
- l_0 est la référence à l'instance vide ;
- s_0 est l'état initial de la mémoire ;
- π_0 est la pile vide ;
- I_0 est le contenu initial de l'entrée standard (défini par l'utilisateur) ;
- O_0 est la suite vide ;
- $e_0 = \{nvar_1 \mapsto \textbf{noninit}, ...nvar_n \mapsto \textbf{noninit}\}$ (les $nvar_i$ $(1 \leq i \leq n)$ sont les variables locales de la méthode **main**)

Règles de transitions

- $\{p\}\ nfield = nvar\ \{q\}$

$$< p, l, e, s, \pi, I, O > \rightarrow < q, l, e, s', \pi, I, O >$$

où
- $s' = (update\ nfield\ l\ v\ s)$
- $v = e[\![nvar]\!] \neq \textbf{noninit}$

Cette transition ne modifie que le store. On effectue la fonction *update* si *nvar* est initialisée. Sinon, la transition échoue.

- $\{p\}\ nvar = expr\ \{q\}$

$$< p, l, e, s, \pi, I, O > \rightarrow < q, l, e', s, \pi, I, O >$$

où
- $e' = e[nvar/v]$
- $v = \mathcal{V}[\![expr]\!]\ e\ s\ l \neq \textbf{erreur}$

- $\{p\}\ nvar = \textbf{new}\ nclass()\ \{q\}$

$$< p, l, e, s, \pi, I, O > \rightarrow < q, l, e', s', \pi, I, O >$$

où
- $e' = e[nvar/l']$
- $l' = newLoc(s)$
- $f = newIns(nclass)$
- $s' = s[l'/ < nclass, f >]$

Lorsqu'on crée une instance, on ajoute une location via la fonction *newloc*. Cette location est ajoutée dans le domaine du store. L'instance qui lui est associée porte le nom *nclass* et a tous ses champs initialisés à la valeur **null**. De plus, cette nouvelle location devient l'image de la variable *nvar* dans l'environnement (i.e *nvar* pointe vers l'instance nouvellement créée).

- $\{p\}$ **if** *cond* **then** q **else** r

$$< p, l, e, s, \pi, I, O > \rightarrow < t, l, e, s, \pi, I, O >$$

où
- $s = q$ si $b =$ **true**
  $s = r$ si $b =$ **false**
- $b = \mathcal{B}[\![cond]\!]\ e\ s\ l \neq$ **erreur**

- $\{p\}$ **return** *nvar*

$$< p, l, e, s, \pi, I, O > \rightarrow < r, l', e', s, \pi', I, O >$$

où
- $\pi = \boxed{\frac{<q, l', e''>}{\pi'}}$
- $findInstr(q) = [\![\{q\} nvar' = cible.nmethod(...)\{r\}]\!]$
- $e' = e''[nvar'/v]$
- $v = e[\![nvar]\!] \neq$ **noninit**

Une instruction "**return** *nvar*" se trouve toujours à la fin d'une méthode. Lorsque cette instruction est exécutée, il s'agit donc de revenir à l'instruction qui a invoqué cette méthode.Le point de programme de cette instruction se trouve au sommet de la pile des appels suspendus. Grâce à la fonction $findInstr$ nous pouvons retrouver l'instruction de ce point de programme. Cette instruction, toujours de la forme $nvar' = cible.nmethod(...)$ donne à $nvar'$ la valeur de *nvar*. De plus, la location change si la méthode appelante n'est pas dans la même instance que la méthode appelée.

- $\{p\}$ $nvar_1$ =**toString**($nvar_2$) $\{q\}$

$$< p, l, e, s, \pi, I, O > \rightarrow < r, l, e', s, \pi, I, O >$$

où
- $e' = e[nvar_1/str]$
- str est la représentation de $(e[\![nvar_2]\!])$ sous la forme de string.

La représentation d'une valeur sous forme de string n'est pas définie si c'est **noninit**. Elle dépend de s, si c'est une location.

- $\{p\}$ **read**($nvar$)$\{q\}$

$$< p, l, e, s, \pi, I, O > \rightarrow < r, l, e', s', \pi, I', O >$$

où
- $e' = e[nvar/l']$
- $I = str.I'$
- $l' = newLoc(s)$
- $s' = s[l' / <\textbf{String}, str>]$

Cette instruction lit un string rentré par l'utilisateur. Il s'agit du premier string de l'input et, une fois lu, ce string est retiré de l'input. Une location est créée et une instance de string (de valeur *str*) lui est associée. De plus, dans l'environnement, la valeur de *nvar* devient cette nouvelle location.

- $\{p\}$ **write**($nvar$) $\{q\}$

$$< p, l, e, s, \pi, I, O > \rightarrow < r, l, e, s, \pi, I, O' >$$

où
- $O' = Ostr$ si $s(e[\![nvar]\!]) =< \textbf{String}, str >$
- erreur sinon

- $\{p\}$ $nvar = cible.nmethod(nvar_1 ... nvar_m)$ $\{q\}$

$$< p, l, e, s, \pi, I, O > \rightarrow < r, l', e', s, \pi', I, O >$$

où
- $\mathcal{T}[\![cible]\!]\ e\ s\ l =< nclass', l' >$
- $findMeth(nclass', nmethod) = declmethod$ avec
  $declmethod = [$ **method** $nmethod\ r$
  $type_1\ nparam_1 \ldots\ type_m\ nparam_m$
  $type'_1 nvar_1\ \ldots\ type'_n\ nvar'_n$
  $instr^*\ ]$
- $v_i = e[\![nvar_i]\!] \neq$ **noninit** $(i = 1...m)$
- $e' = \{nparam_1 \mapsto v_1, ..., nparam_m \mapsto v_m, nvar_1 \mapsto$ **noninit**$, ..., nvar_n \mapsto$ **noninit**$\}$
- $\pi' = \boxed{\begin{array}{|c|}\hline <p,l,e> \\ \hline \pi \\ \hline \end{array}}$

Il s'agit tout d'abord de trouver la méthode $nmethod(nvar_1 ... nvar_m)$. Pour cela, on recherche le type de *cible* et, si c'est bien une instance de classe, on y cherche la méthode via la fonction $findMeth$. On sauve ensuite au sommet de la pile le point de programme, l'environnement et la location. On peut alors passer à la première

instruction de la méthode *nmethod*(...). Dans l'environnement, les valeurs des paramètres formels sont les valeurs des paramètres de l'appel (si $\neq$ **noninit**) et les variables locales sont non initialisées (d'où, leur valeur est **noninit**). Enfin, si la "nouvelle" méthode n'est pas dans la même instance que la méthode appelante (i.e si *cible* $\neq$**this**) la location devient celle de l'instance de la méthode appelée.

- $\{p\}$ **return**

$$< p, l_0, e, s, \pi_0, I, O > \rightarrow < O >$$

où
- l_0 est la référence de l'instance vide
- π_0 est la pile vide

L'état O correspond au résultat final du programme c'est-à-dire à un état final correct. L'état final peut cependant être un état erreur qui serait un état "normal" à partir duquel aucune transition correcte n'est possible.

Chapitre 8

Définition des domaines abstraits et des fonctions de concrétisation

La première étape, qui consiste à définir au niveau syntaxique comme au niveau sémantique notre langage, est maintenant terminée. Pour rappel, l'objectif suivant est de réaliser l'exécution abstraite du programme considéré comme une suite de transitions entre des états abstraits.

Pour ce faire, l'idée est d'abstraire le domaine des valeurs $\mathbb{V}al$, par un ensemble de valeurs abstraites, un domaine primitif abstrait A.

Ce domaine n'est défini qu'ultérieurement, juste avant d'exécuter l'algorithme du point fixe dont il est le "paramètre". A ce niveau, A reste donc un domaine *générique* et inconnu. En réalité, quelques caractéristiques de A sont connues. Premièrement, par définition, A est muni d'une relation d'ordre et d'au moins une borne supérieure. Deuxièmement, nous avons décidé qu'il s'agirait d'un ensemble *fini*. Enfin, l'objectif de notre travail implique que les propriétés des valeurs concrètes que A doit représenter portent sur leur type. Sachant cela, le terme le plus approprié pour nommer les valeurs du domaines A nous semble être les *types abstraits*. Ces types abstraits ce domaine doivent être liés à l'ensemble des types $\mathbb{T}ype$ par une fonction de concrétisation $Cc : A \rightarrow \mathcal{P}(\mathbb{T}ype + \{\textbf{noninit}\} + \{\textbf{null}\})$

Lorsque nous instancions ce domaine A dans la section 2.5.1 nous le faisons de plusieurs façons, en fonction d'un certain compromis entre efficacité et précision de la vérification des types dynamiques. Nous effectuons alors des tests pour différents programmes et comparons les résultats selon le domaine A choisi.

C'est en fonction du paramètre A que nous définissons les domaines abstraits. La structure de ces domaines abstraits est fortement inspirée de celle des domaines sémantiques (concrets), le but étant juste de simplifier ceux-ci. Nous avons choisi une représentation

des états abstraits telle que pour tout programme, ces domaines abstraits soient finis.

Ci-après sont donc définies les différentes composantes d'un état abstrait. Nous avons le store abstrait, l'instance abstraite, l'environnement abstrait et la pile abstraite. Ensuite est défini l'état abstrait en lui-même.

Enfin, les liens entre les domaines sémantiques concrets et ces domaines abstraits sont définis via des fonctions de concrétisation. Cela nous permet finalement de définir le lien entre un état concret et l'état abstrait lui correspondant.

8.1 Domaines abstraits

- A : domaine primitif abstrait ; ensemble des types abstraits

  Un certain nombre d'hypothèses sont posées sur ce domaine primitif abstrait :
  - Soit *type* un type concret (i.e **int**, nclass, **String**), on suppose qu'il existe une fonction générique

    $$abs : \mathbb{T}ype + \{\textbf{noninit}\} + \{\textbf{null}\} \rightarrow A$$

    telle que $abs(t)$ désigne l'abstraction de t.
    La fonction de concrétisation est définie à partir de la fonction d'abstraction, comme suit :

    $$Cc : A \rightarrow \mathbb{T}ype + \{\textbf{noninit}\} + \{\textbf{null}\}$$

    telle que $t \in Cc(abs(t))$.

  - Soient t_a, u_a des types abstraits, on suppose qu'il existe une fonction

    $$sup : A \times A \rightarrow A$$

    telle que $sup(t_a, u_a)$, notée $t_a \sqcup u_a$, est la borne supérieure entre les types t_a et u_a, en supposant qu'il y a une relation d'ordre partielle $\preceq_a$ entre les types abstraits. On suppose en plus que $Cc(t_a) \cup Cc(u_a)$ est inclue dans $Cc(t_a \sqcup u_a)$
  - La borne supérieure entre tous les éléments du domaine est noté $\top$.

- $\mathbb{S}tore_a = \mathbb{N}class \rightarrow (\mathbb{N}field \rightarrow A)$: le store abstrait

  Rappel du store concret : $\mathbb{S}tore = \mathbb{L}oc \rightharpoonup \mathbb{I}nst$

  Le store abstrait est une fonction qui associe à chaque classe déjà instanciée l'évaluation de ses champs. Par rapport au store concret (voir 2.2.3.2), la notion de location

disparaît. Cela permet au store abstrait d'avoir un domaine de définition fini. En contrepartie, ce choix implique que, pour toute classe déjà instanciée, seules restes des approximations des types de ses champs. Intuitivement, ces approximations s'obtiennent en calculant la borne supérieure entre les types abstraits des champs liés à chaque instance d'une même classe.

- $\mathbb{I}nst_a = \mathbb{N}class \times (\mathbb{N}field \to A)$: l'instance abstraite

  Rappel de l'instance concrète :

  $$\mathbb{I}nst = \mathbb{N}class \times (\mathbb{N}field \rightharpoonup \mathbb{V}al) + \{\mathbb{S}tring\} \times \mathbb{C}har^*$$

  L'abstraction des instances concrètes de classe consiste à abstraire les valeurs des champs. L'abstraction des instances de string est tout simplement l'abstraction du type **String**. C'est donc un type abstrait (i.e qui se trouve dans A).

- $\mathbb{E}nv_a = (\mathbb{N}param + \mathbb{N}var \to A)$ **:** l'environnement abstrait

  Rappel de l'environnement concret :

  $$\mathbb{E}nv = (\mathbb{N}param + \mathbb{N}var \rightharpoonup \mathbb{V}al + \{noninit\})$$

  L'environnement abstrait est une fonction associant aux paramètres et variables locaux un type abstrait.

- $\mathbb{P}ile_a = ((\mathbb{P}ts \times \mathbb{I}nst_a \times \mathbb{E}nv_a) \rightharpoonup \mathcal{P}(\mathbb{P}ts \times \mathbb{I}nst_a \times \mathbb{E}nv_a)) \times (\mathbb{P}ts \times \mathbb{I}nst_a \times \mathbb{E}nv_a)$: la pile abstraite

  Rappel de la pile concrète : $\mathbb{P}ile = (\mathbb{P}ts \times \mathbb{L}oc \times \mathbb{E}nv)^*$

  Les éléments de la pile des appels suspendus que nous comptons garder est une abstraction du sommet de la pile ainsi qu'une abstraction de l'"ordre" de la pile concrète. Bien sûr, nous admettons qu'il y ait, au niveau abstrait indétermination. C'est ainsi que pour chaque élément de la pile, il y a un ensemble de précédants possibles.

  La relation de succession est exprimée par la projection 1 du produit cartésien tandis que le sommet de la pile abstraite est lui représenté par la projection 2.

  Notons que tous les paramètres de la pile abstraite sont des domaines finis et que donc le nombre d'états possibles de la pile abstraite pour un programme est fini.

- $\mathbb{E}tat_a = (Pts \times \mathbb{I}nst_a \times \mathbb{E}nv_a \times \mathbb{S}tore_a \times \mathbb{P}ile_a)$: l'état abstrait

  Nous avons choisi d'ignorer dans l'abstraction les input et output d'un programme. Intuitivement, cela signifie que, toute autre chose égale par ailleurs, l'abstraction de deux états concrets ayant un input et/ou un output différent(s) est la même.

8.2 Fonctions de concrétisation

Les fonctions de concrétisation expriment les liens entre les domaines sémantiques (concrets) et les domaines abstraits. Elles servent notamment à prouver les règles de transitions abstraites exprimées dans le chapitre 9.

Ci-après, nous définissons successivement les fonctions de concrétisation $Cc(s_a)$, $Cc(i_a, s_a)$, $Cc(e_a, s_a)$ et $Cc(\pi_a, s_a)$.[1] Les fonctions de concrétisation les plus complexes sont expliquées. La fonction de concrétisation pour les états abstraits est ensuite déduite.

- **Concrétisation du store abstrait**
  $Cc : \mathbb{S}tore_a \rightarrow \mathcal{P}(\mathbb{S}tore)$

$$
\begin{array}{l}
Cc(s_a) = \{\ s | \forall l \in dom(s) : \\
\qquad \text{si } s(l) \text{ est de la forme } < nclass, f >, \\
\qquad \text{alors si } f_a = s_a(nclass), \text{ on a que :} \\
\qquad \forall nfield \in dom(f) : \\
\qquad \text{si } f(nfield) \in \mathbb{Z} \text{ alors } int \in Cc(f_a(nfield)) \\
\qquad \text{si } f(nfield) \in \mathbb{L}oc \text{ alors } f(nfield) = l' \\
\qquad\qquad\qquad \text{avec } l' \in dom(s) \text{ telle que} \\
\qquad\qquad\qquad < nclass', f' > \in Cc(f_a(nfield)) \\
\qquad\qquad\qquad \text{où } < nclass', f' > = s(l') \\
\qquad \text{si } f(nfield) = null \text{ alors } null \in Cc(f_a(nfield)) \\
\qquad \}
\end{array}
$$

  La concrétisation d'un store abstrait est un ensemble de stores. Les domaines de ces stores contiennent un nombre indéterminé de locations. Pour chacun de ces stores :

  - Aucune condition n'est imposée sur les locations telles que $s(l)$ est de la forme $< String, str >$;

[1] Nous devons définir les concrétisations de l'instance abstraite, de l'environnement abstrait et de la pile abstraite en fonction du store abstrait. En effet, dans chacune de ces définitions intervient la concrétisation des types abstraits représentant les classes et les string. Or cette concrétisation se fait par l'intermédiaire du store concret en y insérant les locations.

- Pour les autres locations ($s(l)$ est de la forme $< nclass, f >$), il faut examiner les valeurs des champs. En effet, il est *nécessaire* que le type donné à un champ au niveau concret appartienne à la concrétisation de la valeur abstraite du champ dans $s_a(l)$.

• **Concrétisation de l'instance abstraite (paramétrée par le store abstrait)**
$Cc : (\mathbb{I}nst_a \times \mathbb{S}tore_a) \rightarrow \mathcal{P}(\mathbb{I}nst \times \mathbb{S}tore)$

$$
\begin{aligned}
Cc(i_a, s_a) = \{\ & (< nclass, f >, s) | s \in Cc(s_a) \text{ et } nclass = pr_1(i_a) \\
& \text{et si } f_a = pr_2(i_a), \text{ alors on a que :} \\
& \forall nfield \in dom(f) : \\
& \text{si } f(nfield) \in \mathbb{Z} \text{ alors } int \in Cc(f_a(nfield)) \\
& \text{si } f(nfield) \in \mathbb{L}oc \text{ alors } f(nfield) \in l' \\
& \qquad \text{tel que } l' \in dom(s) \\
& \qquad \text{et } < nclass', f' > \in Cc(f_a(nfield)) \\
& \qquad \text{où } s(l') =< nclass', f' > \\
& \text{si } f(nfield) = null \text{ alors } null \in Cc(f_a(nfield)) \\
& \}
\end{aligned}
$$

Le store abstrait est un paramètre nécessaire dans le cas où l'on veut considérer le champ d'une instance dont la valeur serait une location.

• **Concrétisation de l'environnement abstrait (paramétré par le store abstrait)**

$Cc : (\mathbb{E}nv_a \times \mathbb{S}tore_a) \rightarrow \mathcal{P}(\mathbb{E}nst \times \mathbb{S}tore)$

$$
\begin{aligned}
Cc(e_a, s_a) = \{\ & (e, s) | s \in Cc(s_a) \text{ et } \forall x \in dom(e) : \\
& \text{si } e(x) \in \mathbb{Z} \text{ alors } int \in Cc(e_a(x)) \\
& \text{si } e(x) \in \mathbb{L}oc \text{ alors } e(x) = l \\
& \qquad \text{avec } l \in dom(s) \text{ où } s \in Cc(s_a) \\
& \qquad \text{et } < nclass, f > \in Cc(e_a(x)) \text{ où } s(l) =< nclass, f > \\
& \text{si } e(x) = null \text{ alors } null \in Cc(e_a(x)) \\
& \text{si } e(x) = noninit \text{ alors } noninit \in Cc(e_a(x)) \\
& \}
\end{aligned}
$$

• **Concrétisation de la pile abstraite (paramétrée par le store abstrait)**
$Cc : (\mathbb{P}ile_a \times \mathbb{S}tore_a) \rightarrow \mathcal{P}(\mathbb{P}ile \times \mathbb{S}tore)$

Soit π_a, une pile abstraite ; notons $< p_{a_1}, i_{a_1}, e_{a_1} >$ le sommet de la pile abstraite

i.e :$pr_2(\pi_a)$ et f la fonction qui détermine l'ensemble des précédents possibles dans la pile abstraite (non déterministe) (i.e $pr_1(\pi_a)$).
On a que :

$$
\begin{aligned}
Cc(\pi_a, s_a) = \{\ & (\pi, s) | s \in Cc(s_a) \text{ et } \pi \text{ est de la forme} \\
& < p_1, l_1, e_1 >, < p_2, l_2, e_2 >, \ldots, < p_n, l_n, e_n > \\
& \qquad \text{avec } < p_1, l_1, e_1 >, \text{ le sommet de } \pi. \\
& \text{On a que } p_1 = p_{a_1},\ (s(l_1), s) \in Cc(i_{a_1}, s_a),\ (e_1, s) \in Cc(e_{a_1}, s_a) \text{ et} \\
& \forall < p, i_a, e_a > \in E \text{ où } E = f(p_1, i_{a_1}, e_{a_1}), \\
& \text{on a } (\pi \setminus \{< p_1, l_1, e_1 >\}, s) \in Cc((f, < p, i_a, e_a >), s_a) \\
& \}
\end{aligned}
$$

La définition de cette fonction est récursive. Elle dit que pour chacune des piles π concrètes concrétisant la pile abstraite π_a :
- le sommet $< p_1, l_1, e_1 >$ doit appartenir à la concrétisation du sommet abstrait $< p_{a_1}, i_{a_1}, e_{a_1} >$ de π_a. Pour cela, il faut que chaque élément du triplet formant le sommet appartienne à la concrétisation de l'élément du sommet abstrait lui correspondant.
- la pile concrète, si on lui enlève le sommet, doit appartenir à la concrétisation de toutes les piles abstraites dont le sommet serait un des précédants possibles de $< p_{a_1}, i_{a_1}, e_{a_1} >$.

- **Concrétisation de l'état abstrait**

$(\mathbb{P}ts \times \mathbb{I}nst_a \times \mathbb{E}nv_a \times \mathbb{S}tore_a \times \mathbb{P}ile_a) \rightarrow (\mathbb{P}ts \times \mathbb{L}oc \times \mathbb{E}nv \times \mathbb{S}tore\mathbb{P}ile \times \mathbb{I} \times \mathbb{O})$

$$
\begin{aligned}
Cc((p, i_a, e_a, s_a, \pi_a)) = \{\ & (p, l, e, s, \pi, in, out) |\ s \in Cc(s_a),\ (l, s) \in Cc(i_a, s_a), \\
& (e, s) \in Cc(e_a, s_a),\ (\pi, s) \in Cc(\pi_a, s_a), \\
& \text{et } in, out \in \mathbb{C}har^* \\
& \}
\end{aligned}
$$

Chapitre 9

Sémantique abstraite

Pour pouvoir concevoir l'exécution abstraite des programmes qui nous mènera au point fixe, nous avons besoin d'une sémantique abstraite.

Pour définir cette sémantique abstraite, nous commençons, dans ce chapitre, par définir certaines fonctions sémantiques abstraites. Il s'agit, par exemple, de savoir comment, au niveau abstrait, est évaluée une expression. Ensuite, nous définissons les règles de transitions abstraites, c'est-à-dire que nous décrivons, pour chaque instruction, la règle suivie lorsqu'elle est exécutée sur un état abstrait donné. Enfin, pour clôturer le chapitre, nous prouvons la correction de certaines règles de transitions.

9.1 Fonctions sémantiques abstraites

Mise à jour abstraite d'un champ :

$$update_a : \mathbb{N}field \rightarrow \mathbb{I}nst_a \rightarrow A \rightarrow \mathbb{S}tore_a \rightarrow (\mathbb{S}tore_a \times \mathbb{I}nst_a)$$

$update [\![nfield]\!] \ i_a \ v_a \ s_a = (s'_a, i'_a)$
où $dom(s'_a) = dom(s_a)$ et
soit $nclass = pr_1(i_a)$,
$\forall nclass' \in dom(s_a) \setminus \{nclass\}$, $s'_a(nclass') = s_a(nclass')$ et
$s'_a(nclass) = s_a(nclass)[nfield \setminus sup(v_a, v'_a)]$
où $sup(v_a, v'_a)$ est la borne supérieure entre la nouvelle valeur abstraite v_a et l'ancienne valeur abstraite v'_a du champ.
et $pr_1(i'_a) = pr_1(i_a)$ et $pr_2(i'_a) = pr_2(i_a)[nfield/v_a]$

Lorsqu'on met à jour un champ, l'instance courante i_a et le store s_a sont modifiés. La valeur de l'instance reste la même partout sauf pour le champ *nfield* où la valeur devient v_a.
Le nouveau store est le même que le présentant sauf en $s_a(C)$ où la valeur du champ *nfield* devient la borne supérieure entre l'ancienne valeur et la nouvelle. Le fait de prendre la borne supérieure et non de simplement remplacer la valeur est dû au fait que notre store abstrait est global à tout le programme et non local (i.e différent pour chaque état).

Sémantique des opérateurs

Les opérateurs arithmétiques

Le résultat d'une opération arithmétique peut soit appartenir au domaine primitif A, soit être une erreur, soit appartenir au produit cartésien entre le domaine A et $\{\textbf{erreur}\}$. Le résultat est erreur lorsqu'on est sûr que l'opération est impossible étant donné les types abstraits des opérandes. Si on ne peut pas dire qu'il y a erreur, le résultat est un produit cartésien.
$\mathcal{O} : Aop \rightarrow A \rightarrow A \rightarrow A + \textbf{erreur} + (A \times \textbf{erreur})$

La somme des deux opérandes vaut $abs(int)$ si les valeurs abstraites de ces opérandes spécialisent $abs(int)$. Elle vaut erreur si **int** n'appartient pas à la concrétisation d'une des deux opérandes. Dans les autres cas, il y a indétermination.

$$\begin{aligned}\mathcal{O}[\![+]\!]\, v_{a_1}\ v_{a_2} = {} & abs(int) \text{ si } v_{a_1} \preceq_a abs(int) \text{ et } v_{a_2} \preceq_a abs(int) \\ & \textbf{erreur} \text{ si } \textbf{int} \notin Cc(v_{a_1}) \text{ ou } \textbf{int} \notin Cc(v_{a_2}) \\ & (abs(int), \textbf{erreur} \text{ sinon}\end{aligned}$$

Les autres opérations fonctionnent selon le même principe.

Les opérateurs de comparaison

Le résultat d'une comparaison est toujours, soit une indétermination entre **true** et **false**, soit une erreur, soit une indétermination entre **true**, **false** et **erreur**.
$\mathcal{C} : Cop \rightarrow Val \rightarrow Val \rightarrow \{\textbf{false}\} + \{\textbf{erreur}\} + (\{\textbf{true}\} \times \{\textbf{false}\}) + (\{\textbf{true}\} \times \{\textbf{false}\} \times \{\textbf{erreur}\})$

Les opérateurs d'égalité et d'inégalité sont spéciaux car ils peuvent comparer des éléments de types classe ou **String**. Il s'agit alors d'égalité de pointeurs. Au niveau abstrait, on ne peut jamais assurer que deux classes ou deux string désignent la même instance. En effet, étant donné l'absence de location, toutes les instances d'une même classe sont représentées par la même fonction dans le store. De même,

tous les string sont représentés par le même type abstrait $abs(string)$. On peut cependant être sûr que l'égalité est fausse (ou que l'inégalité est vraie) si les opérandes sont de type classe différent ou si l'un est de type string et pas l'autre.
La comparaison entre entiers est toujours indéterminée.

$$
\begin{array}{lll}
\mathcal{C}[\![==]\!]\ v_{a_1}\ v_{a_2} = & \textbf{true, false} & \underline{\text{si}}\ (v_{a_1} \preceq abs(int)\ \underline{\text{et}}\ v_{a_2} \preceq abs(int)) \\
 & & \underline{\text{ou}}\ (v_{a_1} \preceq abs(String)\ \underline{\text{et}}\ (v_{a_2} \preceq abs(String)) \\
 & & \underline{\text{ou}}\ (v_{a_1} \preceq abs(nclass_1)\ \underline{\text{et}}\ (v_{a_2} \preceq abs(nclass_2) \\
 & & \underline{\text{et}}\ Cc(v_{a_1}) \cap Cc(v_{a_2} \neq \emptyset) \\
 & \textbf{false} & \underline{\text{si}}\ (v_{a_1} \preceq abs(String)\ \underline{\text{et}}\ (v_{a_2} \preceq abs(nclass)) \\
 & & \underline{\text{ou}}\ (v_{a_2} \preceq abs(String)\ \underline{\text{et}}\ (v_{a_1} \preceq abs(nclass)) \\
 & & \underline{\text{ou}}\ (v_{a_1} \preceq abs(nclass_1)\ \underline{\text{et}}\ (v_{a_2} \preceq abs(nclass_2) \\
 & & \underline{\text{et}}\ Cc(v_{a_1}) \cap Cc(v_{a_2} = \emptyset) \\
 & \textbf{erreur} & \underline{\text{si}}\ (v_{a_1} \preceq abs(int)\ \underline{\text{et}}\ \textbf{int} \notin Cc(v_{a_2})) \\
 & & \underline{\text{ou}}\ (v_{a_2} \preceq abs(int)\ \underline{\text{et}}\ \textbf{int} \notin Cc(v_{a_1})) \\
 & & \underline{\text{ou}}((v_{a_1} \preceq abs(nclass)\ \underline{\text{ou}}\ (v_{a_1} \preceq abs(string)) \\
 & & \quad \underline{\text{et}}\ \forall x \in (\mathbb{N}class + \textbf{String}) : x \notin Cc(v_{a_2})) \\
 & & \underline{\text{ou}}\ ((v_{a_2} \preceq abs(nclass)\ \underline{\text{ou}}\ (v_{a_2} \preceq abs(string)) \\
 & & \underline{\text{et}}\ \forall x \in (\mathbb{N}class + \textbf{String}) : x \notin Cc(v_{a_1})) \\
 & \textbf{true, false,erreur} & \underline{\text{sinon}}
\end{array}
$$

Le principe de l'inégalité est similaire. Pour les autres opérateurs de comparaison, seules les comparaisons entre **int** ne peuvent renvoyer un résultat autre que **erreur** (i.e $(true, false)$).

Evaluation abstraite d'une expression

Signature

La fonction d'évaluation d'une expression donne comme résultat un type abstrait, **erreur** (si on est sûr qu'il y a erreur)ou le produit cartésien entre un type abstrait et **erreur** (si on ne sait pas s'il y a erreur).

$$\mathcal{V}_a : \mathbb{E}xpr \rightarrow \mathbb{E}nv_a \rightarrow \mathbb{S}tore_a \rightarrow \mathbb{I}nst_a \rightarrow A + \{erreur\}$$

Les champs

La valeur abstraite d'un champ et celle qui est associée à ce champ dans l'instance courante i_a.

$$\mathcal{V}_a[\![nfield]\!]\ e_a\ s_a\ i_a \quad = f_a(nfield)\ \underline{\text{où}}\ f_a = pr_2(i_a)$$

Les paramètres

$$\mathcal{V}_a[\![nparam]\!]\ e_a\ s_a\ i_a \quad = e_a[\![nparam]\!]\ \underline{\text{si}}\ e_a[\![nparam]\!] \not\succ_a abs(noninit)\ \underline{\text{et}}$$

$$\begin{array}{lr} & e_a [\![nparam]\!] \not\preceq_a abs(noninit) \\ erreur & \underline{\text{si}}\ e_a [\![nparam]\!] \preceq_a abs(noninit) \\ (e_a [\![nvar]\!], erreur) & \underline{\text{sinon}} \end{array}$$

Les variables locales

$$\begin{array}{lll} \mathcal{V}_a [\![nvar]\!]\ e_a\ s_a\ i_a & = e_a [\![nvar]\!] & \underline{\text{si}}\ e_a [\![nvar]\!] \not\succ_a abs(noninit)\ \underline{\text{et}} \\ & & e_a [\![nvar]\!] \not\preceq_a abs(noninit) \\ & erreur & \underline{\text{si}}\ e_a [\![nvar]\!] \preceq_a abs(noninit) \\ & (e_a [\![nvar]\!], erreur) & \underline{\text{sinon}} \end{array}$$

Les opérations arithmétiques

$$\mathcal{V}_a [\![nvar_1\ aop\ nvar_2]\!]\ e_a\ s_a\ i_a = \mathcal{O}_a [\![aop]\!]\ (e_a [\![nvar_1]\!])(e_a [\![nvar_2]\!])$$

Les littéraux

$$\begin{array}{lll} \mathcal{V}_a [\![lit]\!]\ e_a\ s_a\ i_a & = abs(int) & \underline{\text{si}}\ lit \in \mathbb{Z} \\ & abs(String) & \underline{\text{si}}\ lit \in \mathbb{C}har^* \\ & abs(null) & \underline{\text{si}}\ lit = \textbf{null} \end{array}$$

Les expressions castées

On n'accepte le cast au niveau abstrait que si le type abstrait de *nvar* spécialise *abs(nclass)*. Si c'est *abs(nclass)* qui spécialise ("strictement") le type abstrait de *nvar*, alors on ne sait pas dire si le cast est valide.

$$\begin{array}{lll} \mathcal{V}_a [\![(nclass)nvar]\!]\ e_a\ s_a\ i_a & = e_a [\![nvar]\!] & \underline{\text{si}}\ e_a [\![nvar]\!] \preceq_a abs(nclass) \\ & (e_a [\![nvar]\!], erreur) & \\ & \quad \underline{\text{si}}\ e_a [\![nvar]\!] \succ_a abs(nclass) & \\ & erreur\ \underline{\text{sinon}} & \end{array}$$

Le désignateur d'instance this

$$\mathcal{V}_a [\![this]\!]\ e_a\ s_a\ i_a = abs(nclass) \text{ avec } nclass = pr_1(i_a)$$

Transformer un string en entier

$$\begin{array}{lll} \mathcal{V}_a [\![\textbf{toInt}(nvar)]\!]\ e_a\ s_a\ i_a & = abs(int) & \underline{\text{si}}\ e_a [\![nvar]\!] \preceq_a abs(String) \\ & (abs(int), erreur) & \\ & \quad \underline{\text{si}}\ e_a [\![nvar]\!] \succ_a abs(String) & \\ & erreur\ \underline{\text{sinon}} & \end{array}$$

Trouver la longueur d'un string

$$\mathcal{V}_a [\![nvar.\textbf{lenght}()]\!]\ e_a\ s_a\ i_a \quad = abs(int)\ \underline{\text{si}}\ e_a [\![nvar]\!] \preceq_a abs(String)$$
$$(abs(int), erreur)\ \underline{\text{si}}\ e_a [\![nvar]\!] \succ_a abs(String)$$
$$erreur\ \ \underline{\text{sinon}}$$

Comparer la longueur de deux string

$$\mathcal{V}_a [\![nvar_1.\textbf{compareTo}(nvar_2)]\!]\ e_a\ s_a\ i_a =$$
$$abs(int)\ \underline{\text{si}}\ e_a [\![nvar_i]\!] \preceq_a abs(String)\ (i = 1, 2)$$
$$erreur\ \underline{\text{si}}\ e_a [\![nvar_i]\!] \not\preceq_a abs(String)\ \underline{\text{et}}$$
$$e_a [\![nvar_i]\!] \not\succ_a abs(String)$$
$$(abs(int), erreur)\ \underline{\text{sinon}}$$

Evaluation abstraite d'une condition

Signature

$\mathcal{B}_a : Expr \to Env_a \to Store_a \to Inst_a \to \{\textbf{false}\} + \{\textbf{erreur}\} + \{\textbf{true}\} + (\{\textbf{true}\} \times \{\textbf{false}\}) + (\{\textbf{true}\} \times \{\textbf{false}\} \times \{\textbf{erreur}\})$

L'opération instanceOf

Parfois, la valeur de cette opération effectuée sur les domaines abstraits peut être déterminée : c'est le cas si le type abstrait de *nvar* spécialise $abs(class)$ ou inversément.

$$\mathcal{B}_a [\![nvar\ \textbf{instanceof}\ nclass]\!]\ e_a\ s_a\ i_a =$$
$$erreur\ \underline{\text{si}}\ e_a [\![nvar]\!] \not\preceq abs(nclass')$$
$$\underline{\text{ou}}\ e_a(nvar) \not\succ abs(nclass')\ \forall nclass' \in \mathbb{N}class$$
$$false\ \ \underline{\text{si}}\ e_a [\![nvar]\!] \not\preceq abs(nclass)$$
$$\underline{\text{et}}\ \exists nclass'\ (\neq nclass)\ \in \mathbb{N}class\ :\ e_a [\![nvar]\!] \preceq abs(nclass')$$
$$true\ \ \underline{\text{si}}\ e_a [\![nvar]\!] \preceq abs(nclass)$$
$$(true, false, erreur)\ \underline{\text{sinon}}$$

Les opérations de comparaison

$$\mathcal{B}_a [\![nvar_1\ cop\ nvar_2]\!]\ e_a\ s_a\ i_a = \mathcal{C} [\![cop]\!]\ (e_a [\![nvar_1]\!])(e_a [\![nvar_2]\!])$$

L'égalité entre deux string

$$\mathcal{B}_a[nvar_1.\textbf{equals}(nvar_2)]\ e_a\ s_a\ i_a =$$
$$(true, false)\ \underline{\text{si}}\ e_a [\![nvar_1]\!]) = e_a [\![nvar_2]\!] = abs(String)$$
$$erreur\ \underline{\text{si}}\ e_a[nvar_1] \neq abs(String)\ \text{ou}\ e_a[nvar_2] \neq abs(String)$$

9.2 Règles de transition abstraites

L'exécution abstraite d'une instruction d'un programme peut être considérée comme une transition entre un état du programme et un autre. De cette façon, l'exécution abstraite d'un programme est assimilée à une suite de transitions d'état à état démarrant par l'état abstrait initial. Ci-après nous définissons successivement l'état abstrait initial d'un programme et, pour chaque type d'instruction, la règle à suivre par la transition correspondante.

Les règles de transition abstraites sont exprimées selon un modèle similaire à celui des-règles de transition "concrètes". Nous donnons quelques explications pour les transitions qui nous semblent les plus complexes.

Etat initial

C'est l'état $< p_0, i_{a_0}, e_{a_0}, s_{a_0}, \pi_{a_0} >$ où

- p_0 est le point d'entrée de la méthode **main** (i.e le label du programme) ;
- i_{a_0} est la référence à l'instance abstraite vide $< Main, f_\emptyset >$;
- $dom(s_{a_0}) = \emptyset$: s_{a_0} est l'état initial de la mémoire ;
- π_{a_0} est la pile vide ;
- $e_{a_0} = \{nvar_1 \mapsto abs(noninit), ...nvar_n \mapsto abs(noninit)\}$ (les $nvar_i$ $(1 \leq i \leq n)$ sont les variables locales de la méthode **main**.).

Règles de transitions

- $\{p\}$ *nfield* $=$ *nvar* $\{q\}$

$$< p, i_a, e_a, s_a, \pi_a > \rightarrow < q, i'_a, e_a, s'_a, \pi_a >$$

$i'a = pr_2(update\ nfield\ i_a\ v_a\ s_a)$
$s'_a = pr_1(update\ nfield\ i_a\ v_a\ s_a)$ si $abs(noninit) \not\preceq v_a$
où $v_a = e_a [\![nvar]\!]$ et $f(nfield) = e_a [\![nvar]\!]$ avec $f = pr_2(i_a)$

Lorsqu'on affecte une valeur abstraite à un champ, sa valeur dans l'instance abstraite est modifiée tandis que sa valeur dans le store abstrait devient la borne supérieure entre l'ancienne et la nouvelle valeur (cf. $update_a$).

- $\{p\}\ nvar = expr\ \{q\}$

$$< p, i_a, e_a, s_a, \pi_a > \rightarrow < q, i_a, e'_a, s_a, \pi_a >$$

$e'_a = e_a[nvar/v_a]$ et $v_a = \mathcal{V}_a[expr]\ e_a\ s_a\ i_a$.

- $\{p\}\ nvar =$ **new** $nclass()\ \{q\}$

$$< p, i_a, e_a, s_a, \pi_a > \rightarrow < s, i_a, e'_a, s'_a, \pi_a >$$

$e'_a = e[nvar/v_a]$ avec $v_a =< nclass, f_a >$ telle que

$\forall nfield \in dom(f_a)$,
$f_a(nfield) = abs(null)$ si *nfield* est déclaré de type classe ou **String**
$f_a(nfield) = abs(int)$ si *nfield* est déclaré de type **int**

Si $nclass \in dom(s_a)$ alors soit $f'_a = s_a(nclass)$
on a $f''_a = s'_a(nclass)$ où $\forall nfield \in dom(f''_a)$:
si *nfield* est déclaré de type **String** ou classe, $f''_a(nfield) = sup(f'_a(nfield), abs(null))$
si *nfield* est déclaré de type **int**, $f''_a(nfield) = sup(f'_a(nfield), abs(int))$
où $sup(a, b)$ est la borne supérieure entre les types abstraits a et b.
Cette instruction crée une nouvelle instance de *nclass*. Au niveau abstrait, si *nclass* appartient déjà au domaine du store, on change son image en affectant à ses champs la borne supérieure entre leur valeur initiale et $abs(null)$ ou $abs(int)$ selon le type déclaré du champ. Si *nclass* n'appartient pas au domaine du store, on l'y ajoute, en affectant à ses champs les valeurs $abs(null)$ ou $abs(int)$.
La valeur de la variable *nvar* est également changée. Sa valeur est, pour le nouvel état abstrait, le couple $< nclass, f >$ où f affecte aux champs de son domaine les abstraction de **null** ou de **int**.

- $\{p\}$**if** $cond\{q\}$ **else**$\{r\}$

$$< p, i_a, e_a, s_a, \pi_a > \rightarrow < t, i_a, e_a, s_a, \pi_a >$$

$t \in \{q, r\}$
- si $\mathcal{B}_a[cond]\ e_a\ s_a\ i_a = true$ alors $t = q$
- si $\mathcal{B}_a[cond]\ e_a\ s_a\ i_a = false$ alors $t = r$
- si $\mathcal{B}_a[cond]\ e_a\ s_a\ i_a = (true, false)$ alors $t = q$ ou $t = r$, on a une **indétermination**.

- $\{p\}$ **return** $nvar$:

$$< p, i_a, e_a, s_a, \pi_a > \rightarrow < r, i'_a, e'_a, s_a, \pi'_a >$$

$pr_2(\pi'_a) \in f(pr_2(\pi_a))$ avec $f = pr_1(\pi_a)$
$pr_1(\pi'_a) = f \setminus f(pr_2(\pi_a))$
et $pr_2(\pi_a) =< q, i'_a, e''_a >$
où $findInstr(q) = [\{q\} nvar' = cible.nmethod(...)\{r\}]$
et $e'_a = e''_a[nvar'/v_a]$ où $v_a = e_a[nvar]$
Cette instruction, toujours située à la fin d'une méthode, renvoit à la méthode appelante. On élimine donc le sommet de la pile. En réalité, au niveau abstrait, on ne fait que changer le sommet de la pile abstraite, le précédent sommet restant néanmoins dans la pile. En effet, il se peut qu'il corresponde à l'abstraction d'un élément de la pile concrète autre que le sommet.
Notons que cette transition mène souvent à plusieurs états, étant donné que la fonction représentant l'ordre de la pile pointe vers un ensemble de triplets. Un état par élément de cet ensemble doit être traité.

- $\{p\}$ $nvar = cible.nmethod(nvar_1, \ldots, nvar_m)\{q\}$

$$< p, i_a, e_a, s_a, \pi_a > \rightarrow < r, i'_a, e'_a, s_a, \pi'_a >$$

$\forall nparam_i : 1 \leq i \leq m$ tel que $nparam_i \in dom(e'_a) : e'_a [\![nparam_i]\!] = v'_{a_i}$ où $v_{a_i} = \mathcal{V} [\![< nvar_i]\!] i_a\ e_a\ s_a\ \pi_a$,
$\forall nvar'_i : 1 \leq i \leq n$ tel que $nvar'_i \in dom(e'_a) : e'_a [\![nvar'_i]\!] = abs(noninit)$.
$i'_a =< nclass, f_a >$ où $abs(nclass) = \mathcal{T}_a(cible)$ et $f_a = s_a(nclass)$.

$\mathcal{T}$ est défini comme suit :
$\mathcal{T} \rightarrow \mathbb{C}ible \rightarrow \mathbb{I}nst - a \rightarrow \mathbb{S}tore_a \rightarrow \mathbb{E}nv - a \rightarrow A +$ **erreur**
- $\mathcal{T}_a[this]\ i_a\ s_a\ e_a = abs(nclass)$ avec $nclass = pr_1(i_a)$
- $\mathcal{T}_a[super]\ i_a\ s_a\ e_a = abs(nclass')$ avec $nclass' = super(nclass)$ où $nclass = pr_1(i_a)$
- $\mathcal{T}_a[nvar]\ i_a\ s_a\ e_a = e_a [\![nvar]\!]$

Soit $f = pr_1(\pi_a)$;
$pi'_a =< f', sommet >$ où $sommet =< p, i_a, e_a >$ et
$dom(f') = dom(f) \cup sommet$ et
$\forall nvar \in \mathbb{E}tat : nvar \neq sommet : f'(x) = f(x)$ et
si $sommet \in som(f)$ alors
$f'(sommet) = f(sommet) \cup \{E \cup \{pr_2(\pi_a)\}\}, \forall E \subseteq \{f(sommet)\}$

- $\{p\}\ nvar_1 = \textbf{toString}(nvar_2)\{q\}$:

$$< p, i_a, e_a, s_a, \pi_a > \rightarrow < r, i_a, e'_a, s_a, \pi_a >$$

où $e'_a = e_a[nvar_1/abs(String)]$

$\{p\}\ \textbf{read}(nvar)\{q\}$:

$$< p, i_a, e_a, s_a, \pi_a > \rightarrow < q, i_a, e'_a, s'_a, \pi_a >$$

$e'_a = e[nvar/v_a]$ avec $v_a = abs(String)$

- $\{p\}\ \textbf{write}(X)\{q\}$:

$$< p, i_a, e_a, s_a, \pi_a > \rightarrow < q, i_a, e_a, s_a, \pi_a >$$

- $\{p\}$ **return** : état final

9.3 Quelques preuves de correction

Prouver la correction de notre sémantique abstraite revient, en termes plus formels, à démontrer le théorème 2.

Théorème 2
Soient
- σ_{init}, *un état "concret" initial,*
- α_{init}, *un état abstrait tel que*

$$\sigma_{init} \in Cc(\alpha_{init})$$

Si l'exécution de l'organigramme passe par l'état $< p, q >$, *pour cet état initial, il existe un état abstrait* α *tel que*

$$\sigma \in Cc(\alpha) \text{ et } \alpha_{init} \mapsto_* < p, \alpha >$$

Puisque, dans notre cas, à chaque règle concrète correspond exactement une règle abstraite, démontrer ce théorème revient en fait à démontrer que chaque règle abstraite

approxime son pendant concret, i.e. si Tr est une règle concrète et Tr_a son homologue abstrait, Tr_a approxime (correctement) Tr si

$$\forall \alpha, \alpha' \in \mathbb{E}tat_a, Tr_a : \alpha \rightarrow \alpha' \Rightarrow (\forall \sigma \in Cc(\alpha)\ tq\ Tr : \sigma \rightarrow \sigma', \sigma' \in Cc(\alpha')).$$

Il est évident qu'une telle démonstration est relativement fastidieuse. C'est pourquoi nous nous contentons ici de donner, à titre d'exemple, la preuve pour deux règles uniquement. Ces deux règles sont : l'affectation d'une valeur à un champ et la création d'une instance.

9.3.1 Affectation d'une valeur à un champ : $nfield = nvar$

Hypothèses

(1)Soit un état concret $etat_1 =< p, l_1, e_1, s_1, \pi_1, I, O >$, (2)soit l'état $etat_2$ obtenu après la transition Tr, $< q, l_1, e_1, s_2, \pi_1, I, O >$, tel que $s_2 = update\ nfield\ l_1\ v_2\ s_1$; on a $< nclass, f[nfield/v_2] >$ où $< nclass, f >= s(l)$ et $v_2 = e_1[nvar]$.
(3)Soit un état abstrait correspondant à $etat_1$, $etat_{a_1} =< p, i_{a_1}, e_{a_1}, s_{a_1}, \pi_{a_1} >$
(4)Soit un état abstrait obtenu après la transition Tr_a $(nfield = nvar)$, $etat_{a_2} =< q, i_{a_1}, e_{a_1}, s_{a_2}, \pi_{a_1} >$ en respectant la règle définie dans la section 9.2.

Thèse

$etat_2 \in Cc(etat_{a_2})$

Preuve

Par (3), on a que $s_1 \in Cc(s_{a_1})$, $(s_1(l_1), s_1) \in Cc(i_{a_1}, s_{a_1})$, $(e_1, s_1) \in Cc(e_{a_1}, s_{a_1})$ et $(\pi_1, s_1) \in Cc(\pi_{a_1}, s_{a_1})$.

On doit montrer que $s_2 \in Cc(s_{a_2})$, $(s_2(l_1), s_2) \in Cc(i_{a_2}, s_{a_2})$, $(e_1, s_2) \in Cc(e_{a_1}, s_{a_2})$ et $(\pi_1, s_2) \in Cc(\pi_{a_1}, s_{a_2})$.

Par (2), $v_2 = e_1[nvar]$,
par (4), $v_{a_2} = sup(v_{a_1}, e_{a_1}(nvar))$.

Or par "utilité de $\sqcup$", on a $Cc(v_{a_1}) \cup Cc(e_{a_1}(nvar)) \subseteq Cc(v_{a_1} \sqcup e_{a_1})$.

Donc $Cc(e_{a_1}(X)) \subseteq Cc(v_{a_1} \sqcup e_{a_1}(nvar))$;
or,$v_2 \in Cc(e_{a_1} [\![nvar]\!])$ et $v_{a_1} \sqcup e_{a_1} [\![nvar]\!] = v_{a_2}$;

il devient évident que $v_2 \in Cc(v_{a_2})$.

Par définition de la fonction de concrétisation du store, on a bien que $s_2 \in Cc(s_{a_2})$. En effet,
$\forall l \in dom(s_2)$, on a $s_2(l)$ et $s_1(l)$ de la forme $< nclass, f >$ et
si $f_a = s_a(nclass)$,
$\forall nvar \in dom(f) \backslash \{nfield\} : f(x), f_a(x)$ inchangés
et $f(nfield) = v_2$ et $f_a(nfield) = v_{a_2}$ satisfont à la définition.

9.3.2 Création d'une instance : $nvar =$ **new** $nclass()$

Hypothèses :

(1)Soit un état concret $etat_1 =< p, l_1, e_1, s_1, \pi_1, I, O >$;

(2)soit l'état $etat_2$ obtenu après la transition Tr, $< q, l_2, e_2, s_2, \pi_1, I, O >$, tel que $l' = newLoc(s_1)$, $e_2 = e_1[nvar/l']$ et $s_2 = s_1[l'/ < nclass, f >]$ avec $f = newIns(nclass)$.

(3)Soit un état abstrait correspondant à $etat_1$, $etat_{a_1} =< p, i_{a_1}, e_{a_1}, s_{a_1}, \pi_{a_1} >$.

(4)Soit un état abstrait obtenu après la transition Tr_a, $etat_{a_2} =< q, i_{a_1}, e_{a_2}, s_{a_2}, \pi_{a_1} >$.

Thèse :

$etat_2 \in Cc(etat_{a_2})$

Preuve :

Il faut prouver que :
- $s_2 \in Cc(s_{a_2})$ $s_2(l') =< nclass, f >$ avec f telle que
  $\forall nfield \in dom(f_2) : f(nfield) = null$ si $nfield$ est déclaré de type classe ou **String**
  $f(nfield) = 0$ si $nfield$ est déclaré de type **int**
  or, $s_{a_2}(nclass) = f_{a_2}$ avec f_{a_2} telle que
  si $nclass \in dom(s_{a_1})$ alors $\forall nfield \in dom(f_{a_2})$:
  $f_{a_2}(nfield) = abs(null) \sqcup f_{a_1}(nfield)$ si $nfield$ est de type classe ou **String**
  $f_{a_2}(nfield) = abs(int) \sqcup f_{a_1}(nfield)$ si $nfield$ est de type **int**

  sinon $\forall nfield \in dom(f_{a_2})$:
  $f_{a_2}(nfield) = abs(null)$si nfield est de type classe ou **String**
  $f_{a_2}(nfield) = abs(int)$ si nfield est de type **int**

  En utilisant la définition de la fonction de concrétisation du store et l'utilité de $\sqcup$, on

prouve aisément que $s_2 \in Cc(s_{a_2})$.
En effet, $null \in Cc(abs(null))$ et $null \in Cc(abs(null) \sqcup f_{a_1}(nfield)))$ et $int \in Cc(abs(int))$ et $int \in Cc(abs(int \sqcup f_{a_1}(nfield)))$.
Donc, dans tous les cas, $f(nfield) \in Cc(f_{a_2}(nfield))$ et donc $s_2 \in Cc(s_{a_2})$.

- $(e_2, s_2) \in Cc(e_{a_2}, s_{a_2})$
  En effet, en utilisant la fonction de concrétisation de l'environnement, on voit qu'il faut que $s_2 \in Cc(s_{a_2})$, on l'a montré ci-dessus ; et $\forall nvar' \in dom(e_2) \backslash \{nvar\}$, $e_{a_2} [\![nvar']\!]$ est inchangé. Par définition de Tr, $e_2 [\![nvar]\!] = l'$ tel que $l' \in dom(s_2)$; soit $< nclass, f >= s_2(l)$, on a bien que $< nclass, f > \in Cc(e_{a_2} [\![nvar]\!])$.

Chapitre 10

Analyseur

Nous avons maintenant à notre disposition les représentations abstraites des états du programme ainsi que des transitions possibles entre ces états. Ces représentations sont paramétrées par un domaine primitif abstrait A sensé représenter des propriétés sur les types des valeurs.

Dès lors, pour être en mesure de réaliser une exécution abstraite du programme, il ne nous reste plus qu'à définir une instanciation de A.

Sur base de l'instanciation de A et du choix d'un algorithme du point fixe, on peut enfin trouver l'ensemble des α tels que $\alpha_{init} \rightarrow^* \alpha$

Ci-après, nous commençons par discuter intuitivement sur certaines instanciations de A. Ensuite, nous décidons d'un algorithme du point fixe pour terminer par des exécutions de cet algorithme en fonction des différents choix de A.

10.1 Instanciations du domaine primitif abstrait

Tout domaine primitif A est une approximation du domaine concret qu'il représente. Cette approximation doit être la plus proche possible de la réalité. Cependant, elle doit permettre des exécutions plus efficaces que les exécutions concrètes. Le choix d'une instanciation de A se fait sur base d'un compromis entre efficacité et correction.

Pour rappel, tout domaine primitif doit au moins posséder une borne supérieure et une relation d'ordre partielle. De plus, nous nous sommes imposé la contrainte que notre domaine primitif soit un ensemble fini.

Enfin, remarquons que notre choix de représentation du store abstrait implique déjà une perte de précision importante vu la disparition de la notion de location.

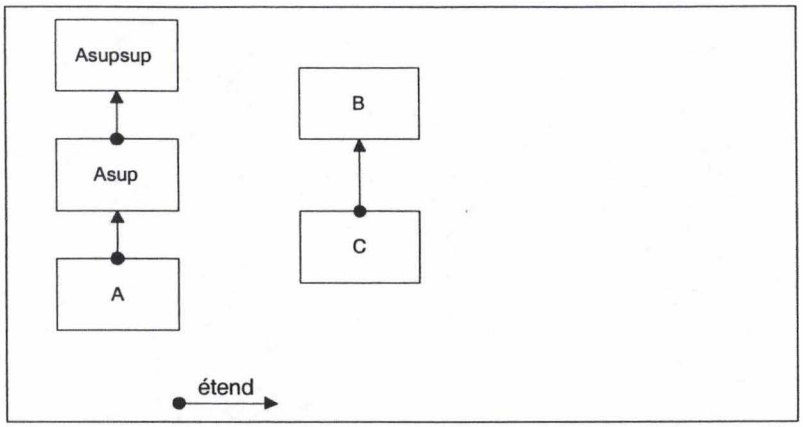


FIG. 10.1 – Exemple de hiérarchie de classes

10.1.1 Le domaine plat

Le choix du domaine plat comme abstraction de l'ensemble des valeurs concrètes $\mathbb{V}ar$ est le plus trivial.

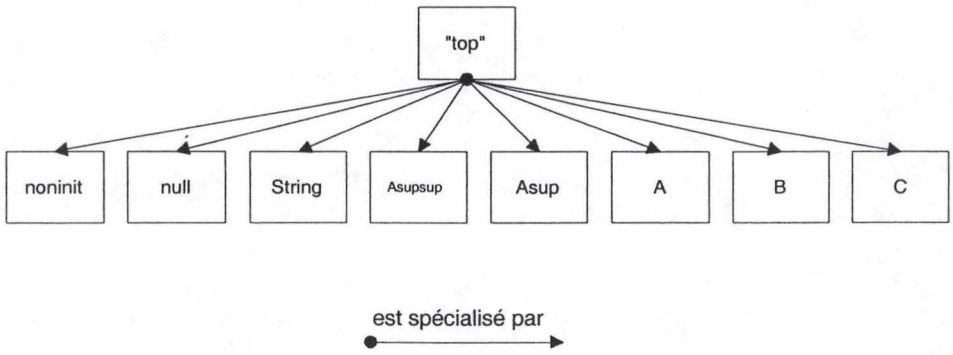
Ce domaine A est défini comme suit (les noms donnés aux types abstraits le sont par simple convention) :

$$A = \mathbb{T}ype + \{null\} + \{noninit\} + \{\top\}$$

où l'ordre partiel est
- $\forall t \in \mathbb{T}ype : t \preceq_a \top$;
- $null \preceq_a \top$;
- $noninit \preceq_a \top$ et

et la borne supérieure est $\top$.

Cela donne, pour l'exemple de la figure 10.1, la structure suivante :



On a la relation suivante :

$$\forall t \in \mathbb{T}ype + \{\textbf{null}\} + \{\textbf{noninit}\} : Cc(abs(t)) = t$$

avec,
- $\forall\ type \in \mathbb{T}ype\ :\ abs(type) = type$;
- $abs(\textbf{null}) = null$;
- $abs(\textbf{noninit}) = noninit$;

Ce choix de domaine a quelques défauts concernant la vérification des types. Nous mettons en évidence, ci-après, deux défauts majeurs de ce domaine.

Tout d'abord, la structure du domaine est explicite, ce domaine implique que toute information concernant la spécialisation entre les classes est perdue.

De plus, considérons par exemple l'instruction `x = new C()` dans un programme quelconque. Définissons $nfield_i$ $(1 \leq i \leq n)$ les champs de type $nclass \in \mathbb{N}class$ et de type **String** déclarés dans C. Les champs $nfield_i$ sont initialisés par défaut à **null**.
Dès lors, par définition de la fonction $update_a$ (section 2.4.2), dès la première instruction $nfield_i = nvar$ du programme, on trouvera dans le store que l'image de $C(nfield_i)$ est $(null \sqcup e_a(nvar))$. Or, en supposant que $e_a(nvar) = nclass$ ou $String$ et sachant que $\forall nclass \in \mathbb{N}class : nclass \sqcup null = \top$ et que $String \sqcup null = \top$, toute information sur le champ $nfield_i$ est d'ores et déja perdue.

De cette façon, nous obtenons trop vite pour les différentes variables du programme des valeurs abstraites $\top$. Ce qui génère des indéterminations partout.

Considérons par exemple, l'instruction `i = x.putval()` où la variable cible x a la valeur abstraite $\top$. On doit dans ce cas traiter l'instruction pour toutes les valeurs possibles de x. Cela implique une indétermination puisque toutes les méthodes *putval()* du programme seront exécutées.

Autre exemple : face à l'instruction `x = (MyString)y`, où y a la valeur abstraite $\top$, on ne peut vérifier si le *cast* est correct. On peut tout juste *supposer* qu'il est correct, et dans ce cas impliquer que dès l'instruction suivante, y est de type $MyString$ [1].

10.1.2 Le cône strict

Ce domaine A est défini comme suit :

$$A = \mathbb{N}class^{\#} + \{int, null, noninit, \top, String\}$$

où

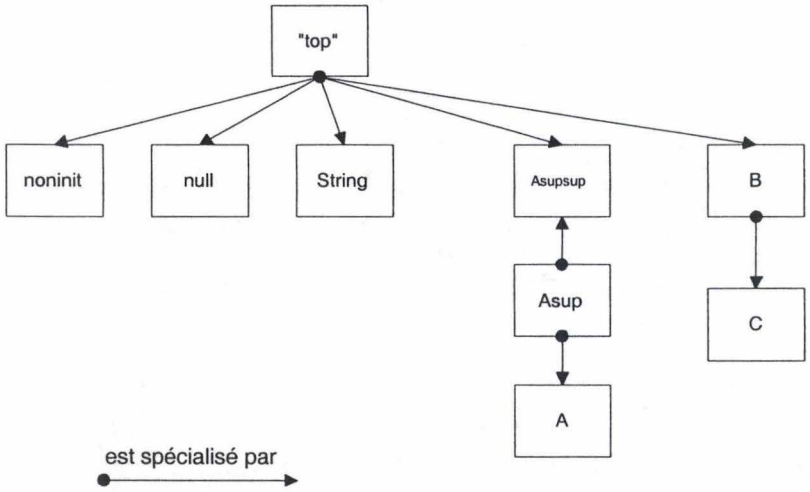
[1] Il s'agit d'une opération de raffinement de l'état abstrait

- $int = abs(\textbf{int})$; $Cc(int) = \textbf{int}$;
- $null = abs(\textbf{null})$; $Cc(null) = \textbf{null}$;
- $noninit = abs(\textbf{noninit})$; $Cc(noninit) = \textbf{noninit}$;
- $String = abs(\textbf{String})$; $Cc(String\#) = \{String\}$;
- $\forall nclass_i \in \mathbb{N}class,\ i \in \mathbb{N} : nclass_i^{\#} \in \mathbb{N}class_i^{\#} \wedge nclass_i^{\#} = abs(nclass_i)$; $Cc(nclass_i^{\#}) = \{nclass_i, nclass_1, ..., nclass_n\}$ où $nclass_1, ..., nclass_n \preceq nclass$.

et où l'ordre partiel est
- $\forall nclass \in \mathbb{N}class^{\#} : t \preceq_a \top$;
- $null \preceq_a \top$;
- $noninit \preceq_a \top$;
- $String \preceq_a \top$;
- $\forall nclass_1, nclass_2 \in \mathbb{N}class : abs(nclass_1) \preceq_a abs(nclass_2) \Leftrightarrow nclass_1 \preceq nclass_2$

et la borne supérieure est $\top$.

Cela donne pour la hiérarchie de classes de la figure 10.1 :



Ce choix de domaine règle le problème de spécialisation posé pour les domaines plats. Cependant, il y a perte de précision dans que l'on abstrait un type classe spécialisé par d'autres classes. Par exemple, dès que l'on abstrait la valeur d'une variable de type *Asup*, on ne sait plus si son type est *A* ou *Asup*.

10.1.3 Le cône

Ce domaine est introduit dans le but de résoudre le problème de l'abstraction de **null** dont la borne supérieure avec l'abstraction d'une classe ou d'un string donne, dans les domaines précédents, $\top$.

Ce domaine est défini comme suit :

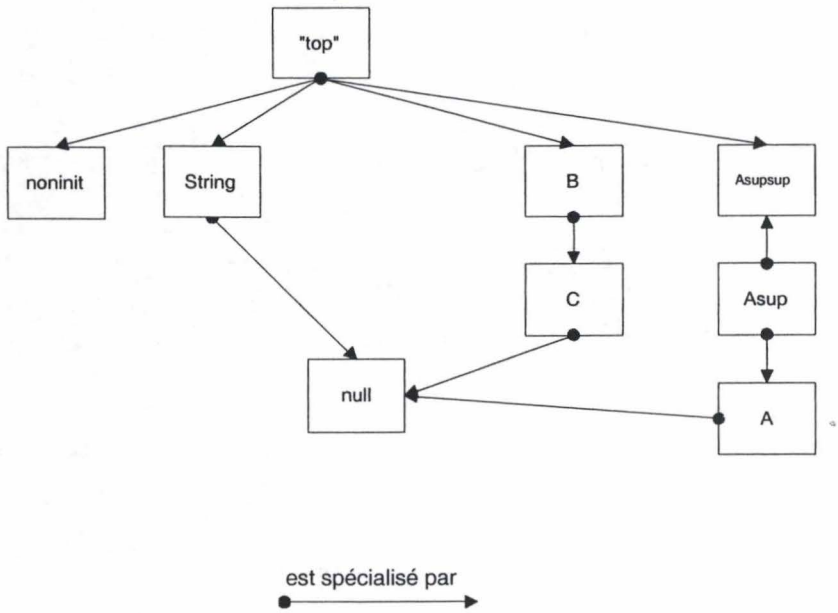
$$A = \mathbb{N}class^{\#} + \{int, null, noninit, \top, String^{\#}\}$$

où
- $int = abs(\textbf{int}) ; Cc(int) = \textbf{int}$;
- $null = abs(\textbf{null}) ; Cc(null) = \textbf{null}$;
- $noninit = abs(\textbf{noninit}) ; Cc(noninit) = \textbf{noninit}$;
- $String^{\#} = abs(\textbf{String}) ; Cc(String\#) = String, null$;
- $\forall nclass_i \in \mathbb{N}class,\ i \in \mathbb{N} : nclass_i^{\#} \in \mathbb{N}class_i^{\#} \wedge nclass_i^{\#} = abs(nclass_i) ; Cc(nclass_i^{\#}) = \{nclass_i, null, nclass_1, ..., nclass_n\}$ où $nclass_1, ..., nclass_n \preceq nclass$.

et la relation d'ordre partiel est
- $\forall nclass^{\#} \in \mathbb{N}class\ :\ nclass^{\#} \preceq_a \top$;
- $\forall nclass^{\#} \in \mathbb{N}class\ :\ null \preceq_a nclass^{\#}$;
- $null \preceq_a String^{\#}$;
- $int \preceq_a \top$;
- $noninit \preceq_a \top$;
- $String^{\#} \preceq_a \top$;
- $\forall nclass_1, nclass_2 \in \mathbb{N}class : nclass_1^{\#} \preceq_a nclass_2^{\#} \Leftrightarrow nclass_1 \preceq nclass_2$

et la borne supérieure est $\top$.

Pour la hiérarchie de la figure 10.1, la structure de ce domaine est :



On peut constater que dans ce domaine, l'abstraction de **null** spécialise ($\preceq_a$) l'abstraction de **String** et de *nclass* ($\forall$ *nclass* $\in$ $\mathbb{N}$*Nclass*).

De cette manière, on n'a plus les inconvénients du domaine plat ou du cône plat. En effet, $abs(nclass) \sqcup abs(\textbf{null})$ correspond à $nclass^{\#} \sqcup null$ et vaut maintenant $nclass^{\#}$ et non plus $\top$. On gagne donc en précision, au niveau du store.

En contrepartie, on peut perdre de la précision au niveau de l'environnement. Par exemple,

- pour le cône strict, $abs(\textbf{String}) = String$ et donc abstraire une valeur de type **String** ne provoque aucune perte d'information sur les types.
- pour le cône, $abs(\textbf{String}) = String^{\#}$ et, comme $Cc(String^{\#}) = \{String, null\}$, on perd manifestement de la précision : une variable qui a la valeur abstraite $String^{\#}$ peut avoir le type **String** comme elle peut être le pointeur null.

Cette restriction fait qu'on n'est pas toujours à même de tester les erreurs de genre *nullPointerExceptions* en Java.

10.1.4 L'ensemble des parties des types

$$A = \mathcal{P}(\mathbb{T}ype)$$

Dans ce cas-là, on a également une borne supérieure $\top$ et une relation d'ordre partielle.

Intuitivement, ce domaine primitif semble amené à générer un nombre énorme d'états différents et sa précision ne semble n' avoir d'égal que son inefficacité.

10.2 Calcul du point fixe

Un fois le domaine abstrait défini, on peut maintenant utiliser un algorithme nous permettant de trouver l'ensemble des états abstraits α tels que $\alpha_{init} \rightarrow^* \alpha$. Plusieurs possibilités d'algorithme sont envisageables dont principalement deux : l'algorithme *multivariant* et l'algorithme *univariant.*

Notre choix est d'appliquer l'algorithme multivariant défini ci-dessous.

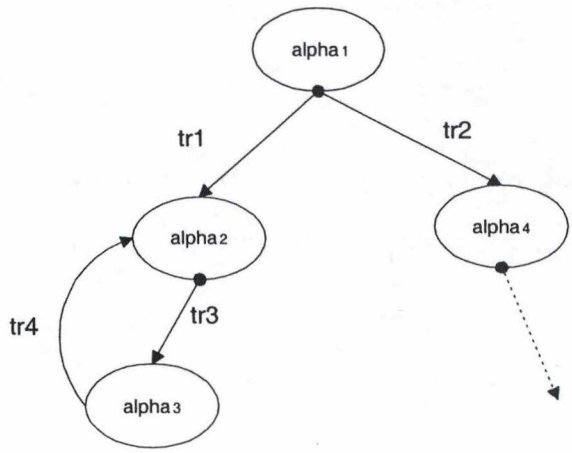
Algorithme 1 *Soient S l'ensemble des états à développer, et R l'ensemble des états déjà développés.*

$S := \{\alpha_0\}$; $R := 0$

tant que $S \neq \{\}$ *faire*
  choisir $\alpha \in S$
  $S := S \setminus \{\alpha\}$
  $R := R \cup \{\alpha\}$
  $S := S \cup (\{\alpha' : \alpha \rightarrow \alpha'\} \setminus R)$

Le principe de cet algorithme est de faire l'exécution abstraite et, pour chaque branche de l'exécution [2], de s'arrêter dès qu'il repasse par un état déjà généré.

Exemple :



Globalement,on peut dire que pour cet exemple, l'algorithme fonctionne comme suit : il commence à l'état α_0 et se trouve déjà face à une indétermination puisque deux transitions (tr_1 et tr_2) partent de cet état. Ensuite, l'algorithme choisit une branche, il effectue par exemple la transition tr_1. Dans ce cas, il atteint successivement les états α_1, α_2 et à nouveau α_1. L'algorithme se rend alors compte, par un test d'égalité, que α_1 a déjà été traité et rompt dans cette direction. Il effectue alors tr_2.

En fait, cet algorithme mémorise tous les états différents générés par l'exécution abstraite du programme.

Un autre algorithme, l'algorithme *univariant*, peut être envisagé. Celui-ci ne mémorise pas tous les états générés. Pout chaque point de programme, il retient uniquement la borne supérieure des états qui y sont générés durant toute l'exécution. Cet algorithme peut engendrer des pertes de décision dans certains domaines abstraits. Il est cependant souvent plus efficace. Le choix de l'algorithme multivariant a été dicté par le simple fait qu'il est plus facile à implémenter. Il est en effet plus facile de tester l'égalité entre deux états que d'en calculer la borne supérieure (notamment pour les piles abstraites).

[2] une exécution abstraite a plusieurs branches lorsqu'on est face à une indétermination.

10.3 Application de l'algorithme multivariant

Nous commençons par montrer et commenter les résultats obtenus par les exécutions abstraites du programme "toto" (Annexe 1) pour différents domaines primitifs (domaine plat, cône strict et cône).

Ensuite, nous détaillons quelques cas subtils d'exécution qui provoquent des indéterminations et nous introduisons les fonctions de raffinement.

10.3.1 Résultats obtenu sur "toto"

Tout d'abord, le tableau 3 reprend les résultats obtenu pour trois domaines primitifs.

Domaine	**nombre d'états générés)**
domaine plat	230
cône strict	230
cône	107

TABLEAU 3 : Résultats des exécutions

Le nombre d'état générés est logiquement inférieur pour le cône. Intuitivement, c'est dû au fait que, dans ce domaine, $\forall nclass \in \mathbb{N}class : abs(nclass) \sqcup abs(\textbf{null}) = abs(nclass)$ et non plus $\top$. En plus de rendre l'exécution plus efficace, ce domaine ajoute de la précision au niveau du store.

Cependant, certaines imprécisions sont plus importantes pour le cône que les autres domaines au niveau de l'environnement (moins de "pointeurs null" détectés).

Notons enfin qu'il n'y a pas de différence entre le cône strict et le domaine plat parce que, dans cette exemple, il n'y a pas d'endroit où la notin d'héritage entre classes est déterminante.

10.3.2 Quelques cas particuliers

10.3.2.1 Les tests abstraits

Un test concret B_c est forcément déterministe : il renvoie soit **true**, soit **false** soit **erreur**. Un test abstrait par contre, peut être indéterminé vu la perte de précision des valeurs abstraites.

Exemple :

Considérons la classe `List` suivante.

```
   class List{
      Cell first;
      int empty(){
           Cell c;
           Cell d;
           int i;
0:         c = first;
1:         d = null;
2:         if(c == d){
3:             c.putVal();
4:             i=0;
           }
           else{
5:             c.getVal();
6:             i=1;
           }
7:         return i;
      }
   }
```

Au niveau concret, la méthode *empty*() contient manifestement une erreur dans le cas où $first = null$. En effet, au point de programme 3, la méthode $putVal()$ est appelée sur une variable de valeur **null** ($d =$ **null** $\wedge\ c = d\ \Rightarrow\ c = null$).

Au niveau abstrait, on ne connaît pas nécéssairement la branche de la condition par laquelle nous devons passer. Souvent même, il y a indétermination et dans ce cas, il faut considérer les deux branches. On peut cependant apporter des précisions au niveau des types grâce aux fonctions de raffinement (*définition 7*).

Définition 6 *Soient les fonctions* $B_a : \Sigma \to \Sigma$ *et* $\bar{B}_a : \Sigma \to \Sigma$*; ce sont des fonctions qui restreignent l'état* $\sigma \in \Sigma$ *en considérant la condition respectivement vraie et fausse.*

Dans l'exemple, supposons qu'au point 2 on ait que $Cc(e_a [\![c]\!])$ *contient null*, si on considère la condition satisfaite, on peut restreindre l'état au point 3 à l'état $B_a(\sigma)$ où $e'_a [\![c]\!] = abs(null)$ (avec e'_a, l'environnement modifié). Dans le cas contraire, l'état $B_a(\sigma)$ au point 5 est tel que $e'_a [\![c]\!] = abs(Cc(e_a [\![c]\!]) \setminus \{null\})$.

Appliquons ce principe aux domaines abstraits du cône strict et du cône :

- **Cône strict**

  Au point de programme 2, on a toujours que $e_a[\![d]\!] = null$. On a également que soit $s_a\ list\ first = null$ ou $s_a\ list\ first = \top$ et donc que $e_a[\![c]\!] = null$ ou $e_a[\![c]\!] = \top$
  - si $e_a[\![c]\!] = null$, il y a détermination : on passe par la branche "then".
  - si $e_a[\![c]\!] = \top$, il y a indétermination :on doit considérer les deux branches et on raffine les états.

- **Cône**

  - si $e_a[\![c]\!] = null$, il y a détermination : on passe par la branche "then".
  - si $e_a[\![c]\!] = String^{\#}$ ou $nclass^{\#}$, il y a indétermination. Il y a cependant raffinement.
  - si $e_a[\![c]\!] = int$ il y a détermination : c'est une erreur et l'exécution est interrompue.
  - si $e_a[\![c]\!] = \top$, il y a indétermination.

10.3.3 Les itérations en interprétation abstraite

Pour une exécution concrète, le principe est de faire l'itération tant que la condition d'arrêt n'est pas satisfaite. Si on voulait vérifier toutes les exécutions possibles, le temps de vérification pourrait être énorme voire infini. Au niveau abstrait, le principe est d'itérer les instructions abstraites de la boucle tant qu'on ne retombe pas sur un état déjà généré. Vu que le nombre d'états abstraits est, dans notre cas, fini, le nombre de passage dans la boucle est nécessairement fini.

Rappelons qu'une boucle *while* consiste, dans ce langage, en une séquence d'instructions labellisées incluant un test.

Ainsi la boucle

```
...
j=1;
i=ls.isEmpty();
while(i!=j){
     x=ls.remove();
     i=ls.isEmpty();
}
...
```

où l'on suppose *ls* initialisé, se traduit dans ce langage en

```
        ...
0       j=1;                1
1       i=ls.isEmpty();     2
2       if i!=j           3 5
3       x=ls.remove();      4
4       i=ls.isEmpty();     2
5       ...
```

L'intérêt de l'interprétation abstraite en est d'autant plus évident ici puisque la méthode appelée dans l'itération, *ls.remove()* consiste en un grand nombre d'instructions avec plusieurs appels de méthodes.

Dans l'exemple, en ne passant qu'une seule fois dans la boucle, les informations sur les types sont obtenues. En effet, lors du second passage au point de programme 2, en se rend compte qu'on revient dans un état déjà exploré.

Remarquons que de façon générale, il est possible de devoir passer plusieurs fois dans la boucle. Ces différents passages peuvent préciser certaines informations via la fonction de raffinement.

10.3.3.1 Le Cast et sa fonction de raffinement

Reprenons l'exemple `x= (MyString) y` où y est de valeur abstraite $\top$.

Nous avons dit précédemment qu'aucune vérification de type ne pouvait être effectuée sur cette expression.En effet, on a aucune information sur y.

L'idée est de dire que si on passe sans provoquer d'erreur dans le cast, alors $e'_a [\![y]\!] \preceq_a abs(MyString)$, où e'_a est l'environnement raffiné suite à l'exécution du cast.

10.3.3.2 Les expressions arithmétiques

L'évaluation d'une expression arithmétique peut également provoquer un raffinement de l'état. Si l'on considère par exemple l'instruction `y = x * 2` où x a la valeur abstraite $\top$, L'exécution de l'instruction suppose que $e'_a [\![x]\!] \preceq_a abs(int)$ (où e'_a est le nouvel environnement).

10.3.3.3 Les appels de méthodes

Le même principe de raffinement est applicable lors des appels de méthode (*cible.nmethod*()). Dans ce cas, si, par exemple, le type abstrait de la variable *cible* est $\top$, il est affiné lorsque

l'on exécute l'instruction.

Conclusion

L'objectif de la première partie de ce mémoire était de contribuer à la réalisation d'un prototype d'analyseur statique des types d'un sous-ensemble de Java. Pour ce faire, nous avons conçu et implémenté un parser de VTF simplifié et une traduction de ce langage vers une syntaxe abstraite appelée SAP. Cette étape est fondamentale dans le cadre du projet global car la syntaxe SAP peut être considérée comme une syntaxe abstraite capable de représenter des programmes sous une forme plus simple plus explicite que VTF simplifié. SAP se prête donc plus aisément à des analyses statiques de types basées sur la théorie de l'interprétation abstraite. La traduction est une phase essentielle dans l'élaboration d'un prototype puisqu'il permet d'effectuer des tests réalistes sur un sous-ensemble raisonnable de Java.

Notons que le sous-langage est évidemment susceptible d'être étendu, et que la façon dont nous avons implémenté le traducteur, en gardant un niveau d'abstraction important, n'entrave pas une telle extension.

Cependant, notre traducteur a quelques défauts. Tout d'abord, si en VTF simplifié, des labels sont syntaxiquement possibles (afin de pouvoir identifier certaines lignes de code), il aurait été intéressant d'établir une correspondance entre ces labels et ceux de SAP. Autre défaut, l'absence de méthode principale en VTF simplifié qui nous oblige a adopter une convention peu intuitive consistant à nommer un méthode du nom **main** dans n'importe quelle classe du programme.

Cependant, concevoir et implémenter une sémantique abstraite sur un langage tel que SAP dépasse le cadre de notre mémoire. C'est pourquoi, dans la deuxième partie, nous avons défini un sous-langage de SAP sur lequel nous avons réalisé notre analyseur statique. Certes, ce langage est considérablement restreint et les domaines abstraits considérés ne sont certainement pas suffisamment expressifs. Néanmoins, quelques tests significatifs sont réalisés et nous permettent d'obtenir une intuition à propos des domaines primitifs implémentés.

D'une part, il serait intéressant de réutiliser nos domaines abstraits pour la réalisation d'un analyseur statique propre à SAP. D'autre part, on pourrait considérer des domaines plus complexes, capables de conserver plus d'informations. Par exemple, il est évident que

l'on pourrait garder plus d'information concernant les instances comme dans, par exemple, [PLCC01], étant donné que nos domaines abstraits intègrent toutes les instances d'une même classe au même endroit.

Bibliographie

[AH87] S. Abramsky and C. Hankin. An introduction to abstract interpretation. In S. Abramsky and C. Hankin, editors, *Abstract Interpretation of Declarative Languages*, chapter 1, pages 9–31. Ellis Horwood Limited, 1987.

[CC77] P. Cousot and R. Cousot. Abstract interpretation : A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of Fourth ACM Symposium on Programming Languages (POPL'77)*, pages 238–252, Los Angeles, California, January 1977.

[CDG96] C. Chambers, J. Dean, and D. Grove. Whole-Program Optimization of Object-Oriented Languages. Technical Report 96–06–02, Department of Computer Science and Engineering, University of Washington, Box 352350, Seattle, Washington 98195–2350 USA, June 1996.

[GJG96] J. Gosling, B. Joy, and Steele G. Java Language Specification. Java Series. Addison-Wesley, 1996.

[LC92] B. Le Charlier. L'Analyse Statique des Programmes par Interprétation Abstraite. *Nouvelles de la Science et des Technologies*, 9(4) :19–25, 1992.

[LC99a] B. Le Charlier. Définition du Langage `Vas-T'y-Frotte`. Notes de cours, 1999.

[LC99b] B. Le Charlier. Interprétation Abstraite. Institut d'Informatique, Université de Namur, Belgique, 1999. Notes de cours.

[NNH99] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis.* Springer-Verlag, 1999.

[PLCC01] I. Pollet, B. Le Charlier, and A. Cortesi. Distinctness and Sharing Domains for Static Analysis of Java Programs. In *Proceedings of (ECOOP'01)*, number 2072 in Lecture Notes in Computer Science, pages 77–98. Springer Verlag, 2001.

[Pol99] I. Pollet. Sémantiques opérationnelles et domaines abstraits pour l'analyse statique de Java. DEA thesis, University of Namur, Belgium, September 1999.

[Sch00] P-Y. Schobbens. Syntaxe et Sémantique. Institut d'Informatique, Université de Namur, Belgique, 2000. Notes de cours.

[WM94] R. Wilhem and D. Maurer. Les compilateurs : théorie, construction, génération. Masson, 1994.

[Y.99] Deville Y. Introduction à la calculabilité. Institut d'Informatique, Université de Namur, Belgique, 1999. Notes de cours.

Annexes

Programme "toto"

Note : il s'agit d'un peudo-code du langage considéré (points de programme implicites)

```
class T{}

class MyString extends T{
    String v;
    int putVal(String s){
        String t;
        int i;
        t=s;
        v=t;
        i=0;
        return i;
    }
    String getVal(){
        String s;
        s=v;
        return s;
    }
}

class Cell{
    T v;
    Cell next ;
    int putVal(T y, Cell suiv){
        T x;
        Cell c;
        int i;
        x =y;
        c=suiv;
```

```
            v=x;
            next = c;
            i = 0;
            return i;
        }
        T getVal(){
            T x;
            x=v;
            return x;
        }
        Cell getNext(){
            Cell c;
            c=next;
            return c;
        }
    }

    class List{
        Cell first;
        int add(T v){
            Cell l;
            T x;
            int i;
            Cell c;
            x=v;
            c=first;
            l=new Cell();
            i=l.putVal(x,c);
            first=l;
            i=0;
            return i;
        }

        int empty(){
            Cell c;
            Cell d;
            int i;
            c = first;
            d=null;
            if(c==d)
            i=1;
            else i=0;
```

```
        return i;
    }

    T remove(){
        T r;
        Cell c;
        c =first;
        r = c.getVal();
        c =c.getNext();
        first =c;
        return r;
    }
}

class ListOfStrings extends List {
    int add(String v){
        MyString mys;
        String s;
        int i;
        s=v;
        mys = new MyString();
        i = mys.putVal(s);
        i = super.add(mys);
        i=0;
        return i;
    }
}

class Main{
    void main(){
        ListOfStrings ls;
        String s;
        String t;
        int i;
        MyString mys;
        T x;
        ls = new ListOfStrings();
        s = "un string svp!";
        write(s);
        read(s);
        i = ls.add(s);
        s = "un entier, svp!";
```

```
            write(s);
            read(i);
            String toto;
            toto="";
            s = toto+i;
            i =ls.add(s);
            s = "l'entier:";
            x = ls.remove();
            mys = (MyString)x;
            t = mys.getVal();
            s = s+t;
            write(s);
            s = "le string:";
            x = ls.remove();
            mys = (MyString)x;
            t = mys.getVal();
            s = s+t;
            write(s);
        }
    }
```